ザ8086ブック［新装版］

16ビット・マイクロプロセッサ8086・8088の使い方

ラッセル・レクター／ジョージ・アレクシー共著　吉川敏則訳

秋葉出版

本書の特色

●ハードウェア

8086チップの構成，タイミング，さらに設計について詳細に解説。

●プログラミング

適切なプログラミング技法の解説とともに，完全な8086命令セットを示す。

●インターフェイス

すべての種類の素子のインターフェイスについての技法と仕様の解説。

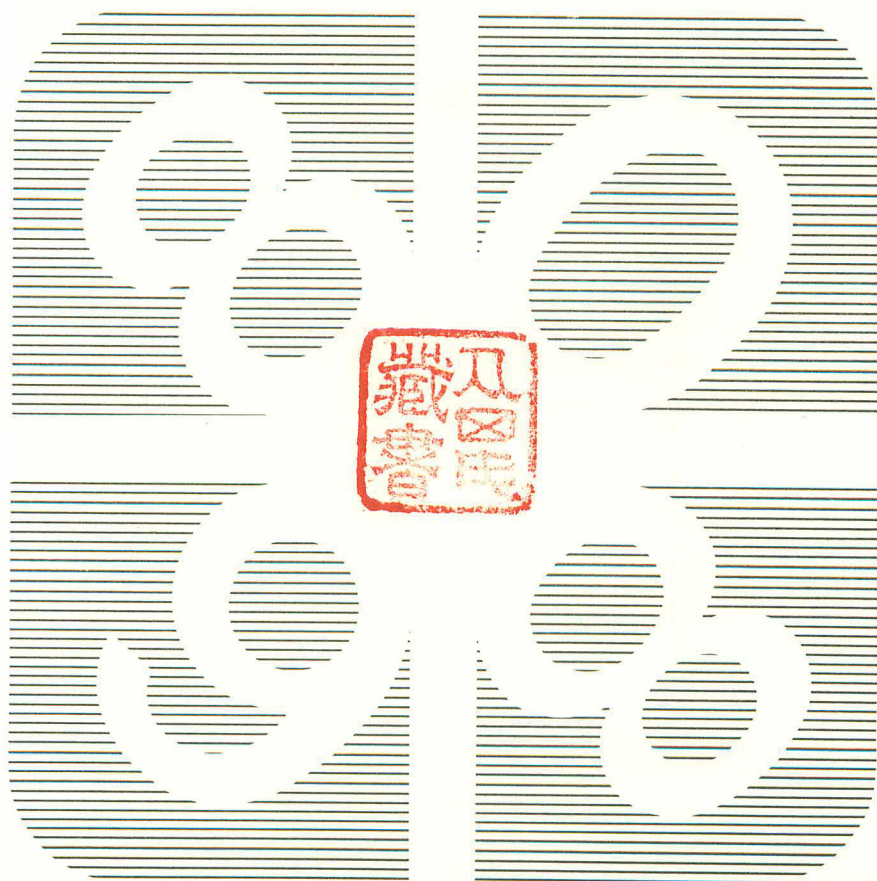
●アプリケーション

8086の特殊な特徴を客観的に示すと共に，Multibusとマルチ CPU の構成についても詳述。

1987年4月25日.
有隣堂にて.

ザ 8086ブック

16ビット・マイクロプロセッサ8086・8088の使い方

ラッセル・レクター／ジョージ・アレクシー 共著　吉川敏則 訳

秋葉出版

The 8086 Book

by Rusell Rector / George Alexy

著者の序文

この本は，一般的なプログラミングの概念と実施，8086マイクロプロセッサとそのアセンブリ言語，8086マイクロプロセッサを用いたロジック・デザインの3つの話題に焦点を置いている．一般的なプログラミングの概念と実施の議論は，どのマイクロプロセッサにも関連している．しかし，この本の残りは8086に特有の部分である．それだけで，この本は8086を用いるためのテキストとなる．

8086マイクロプロセッサのプライム・ソースは，次のものである．

INTEL CORPORATION
3065 Bowers Avenue
Santa Clara, California 95051

一般的なプログラミングの概念と実施の解説は，プログラマとコンピュータとの間の関係を調べることから始まる．というのは，これが結局はデザイン・プロジェクトの本質を決定するものだからである．他のプログラマがアセンブリ言語あるいはより高いレベルの言語でプログラミングを行なっているときに，どうして機械語を用いて行なうプログラマがいるだろうか．異なるタイプの応用には異なるタイプのプログラミングが要求される．いずれの場合においても，良いプログラミングの実施は洗練される必要がある．この目的を成就するために，一連の規則について述べ，プログラミングのプロジェクトを説明するために2つの例を用いる．

8086マイクロプロセッサ自身の記述は，アセンブリ言語のプログラミングとハードウエアの設計を含んでいる．

アセンブリ言語のプログラマに対しては，8086のCPU構成とマイクロプロセッサのアセンブリ言語の命令セットについて，詳細に述べている．

ハードウエアの設計者に対しては，一般にマイクロプロセッサへ入力されるかあるいは，マイクロプロセッサから出力されるすべての信号に関するタイミングとバスの要点について述べている．シングルバスとマルチバスの構成も含まれている．標準のIntel Multibusについても詳細に記述してある．

この本が読者に仮定している知識

An Introduction to Microcomputers: Volume 1 — Basic Concepts, 2nd Revision, by A. Osborne, Osborne/McGraw-Hill, 1980 に述べられている一般的なマイクロプロセ

ッサの概念の実用的な知識や考え方を読者がもっていることを，この本では仮定している．したがって，２進数の算術演算，バッファ，あるいはCPU構成の原理などの基本的要素は，この本に含まれていない．

8086マイクロプロセッサとその直接のサポートの部分については，この本で詳細に取り上げてある．8089 I/O プロセッサにも触れているが，詳しくは述べていない．この詳細な記述については，The 8089 I/O Processor Handbook, by A. Osborne, Osborne/McGraw-Hill, 1980 を参照されたい．

日本語版刊行に際して

1970年代初期に紹介されて以来，マイクロプロセッサが日常生活の中で占める割合は，ますます大きくなってきている．自動車を運転するときも，ゲームマシンで遊ぶときも，われわれは，この強力なプロセッサとインターフェイスし，利用しているわけである．コンピュータをベースにした製品なくしては，各人が日常生活をおくれなくなる日も近いであろう．このような製品を使うことによって，人類は単純労働から解放され，社会を改善するための創造的な活動に従事することができるようになる．

新世代のマイクロプロセッサ技術が出現するたびに，このゴールが近くなってくる．最新の進歩といえば，16ビット・マイクロプロセッサの出現であろう．16ビット・マイクロプロセッサは，従来の8ビット・マイクロプロセッサに比べて，処理速度とデータ量が数倍になっている．この世代の中で最もポピュラーなCPUは8086で，前世代の8ビット・マイクロプロセッサ8080に比べて，処理能力が10倍，メモリ・アドレス領域が16倍になっている．その高度なコンピュータ・アーキテクチャのために，8086ファミリーは，色々なアプリケーション分野で非常にポピュラーになっている．高速，高精度の浮動小数点演算という機能を8087演算プロセッサで追加したり，高度なI/Oコントロールを8089 I/Oプロセッサで追加したりできるという，そのユニークなアーキテクチャは，8086ファミリーを，さらに広範で，高度なアプリケーションに適合させた．

また，そのフレキシブルなアーキテクチャのゆえに，他のどの8ビット・プロセッサに比べても，2倍以上のパフォーマンスを有する8ビットの8088から，8086とソフトウエア互換性を保ちながら，6倍のパフォーマンスと，オンチップのメモリ・マネージメントとメモリ・プロテクション機能を有する286まで，そのファミリーは充実している．

特に重要なのは，8086がパーソナル・コンピュータと科学技術計算の分野で，非常に広範に受け入れられていることである．簡潔なデザインに起因する高いコスト・パフォーマンスと，10社近くあるセカンドソース，そして広範でしかも，さらに増加しつつあるソフトウエアの基盤により，8088と8086は新製品にどんどん採用されている．8088と8086が広く受け入れられているということは，次世代のコンピュータ製品を使う人々のほとんどだれもが，8088や8086ベースの製品を使うことになるであろうということである．したがって，8088と8086のハードウエアとソフトウエアの両面を完全に理解したい，という要求が生まれてくるわけである．

この8086ブックのゴールと目的は，8086のアーキテクチャとアセンブリ言語およびハー

ドウエア設計のテクニックを詳細に説明することによって，この要求を満たすことである．本書を使用することによって，学生あるいは8088，8086のユーザは，この2つのプロセッサに関する知識を，迅速かつ効率的に得ることができ，新たに得た知識をハードウエアとソフトウエアの開発に適用することができる．なお本書は，読者が一般的なマイクロプロセッサの概念を持っていることを前提としている．

本書を精読したなら，読者は世界標準である8088と8086をベースにしたシステムの設計と，プログラミングに参加する資格と自信ができたものと考えてよいだろう．

1982年7月

G. Alexy

訳者のまえがき

私が初めてコンピュータに直接触れることができたのは，今から10年以上前のことになる．当時としてはかなりの規模のコンピュータであったが，今としては時代遅れの機種となっている．それ以来，大型から小型まで，何種類かのコンピュータを使う機会があった．FORTRANやPL/Iなどの高級言語を使用しなかった訳ではないが，私が初めから興味を持っていたのは，主としてアセンブリ言語であったと言える．その理由の1つは，自分自身でコンピュータを動かしてみたい，あるいは操作してみたいといった欲求に駆られたからかもしれない．そして，この欲求を満たしてくれたのは，大型あるいは中型のコンピュータではなく，小型のそれもミニコンと呼ばれる機種であった．中型以上の機種ではそのシステムが複雑で，ユーザの立ち入ることの可能な部分が限られ，あまり興味がわかなかった．あるいは自分の能力の限界を超えていたのかもしれない．

コンピュータは，たとえミニコンと呼ばれるものであっても，決して安価な装置とは言えない．ところが，私が初めてコンピュータに触れることができた頃，ミニコンよりもさらに小型の，マイクロコンピュータが作られていた．正確には，マイクロプロセッサと言うべきであろうが，これがエレクトロニクスの発達の波に乗り，非常な急成長を遂げて現在に至っている．最初は4ビットであったものが，8ビット，さらに16ビットへと発展し，ミニコンとの区別がつかないようにまでなりつつある．また，マイクロプロセッサ自体の価格は非常に安くなり，機器に組み込まれたり，あるいはマイコン（マイクロ・コンピュータ）として，広範囲にわたってマイクロプロセッサが用いられている．

16ビットのマイクロプロセッサは，まだ開発されてから日が浅いが，今後多くの分野で積極的に取り入れられていくことは容易に推測できる．勿論その価格が下がることも要因の1つであろうが，その機能や能力の大きさは非常に魅力がある．

この本には，主として8086について，そのハードウエアとアセンブリ言語による命令の解説が示されている．ただし，著者の序文にも書かれているように，基礎的知識を既に読者がもっていることを仮定しており，また本文中では8086のアセンブリ言語の概要については述べられないままに，プログラムの例が示されている．したがって，この本で初めてアセンブリ言語に接する場合には，多少戸惑いをいだくかもしれない．

また，コンピュータ関係の本に限ったことではないが，この種の本では文章中に専門用語が多く，初心者にはなじみにくいものとなりやすい．本書を訳す際には，専門分野で英語で用いられている用語はできるだけ英語のままで残し，多少でも用語の理解に役立つこ

とを考慮して，注釈としてその訳を付けるようにした．

コンピュータ関係の専門書に少しは慣れているものと思い，この本の翻訳を引き受けてしまったが，いざ始めると自分の英語の貧弱さをまざまざと見せつけられてしまう破目になり，結果的には読者の判断を仰ぐことになってしまった．この本に関して読者の率直な御指摘・御批判をいただければ幸いである．

最後に，翻訳の機会を与えていただいた東京工業大学の当麻喜弘教授ならびに産報出版株式会社の木内雄一氏に心から御礼を申し上げる．

昭和57年7月

吉川　敏則

訳者補足

1980年9月以降，以下のように名称が変更されている．括弧の中は旧名称を示す．

iAPX86/10　(8086)
iAPX88/10　(8088)
iAPX86/20　(8086と8087のシステム)
iAPX88/20　(8088と8087のシステム)

また，iAPX86の拡張として，

iAPX186〔iAPX86にクロック・ジェネレータやDMAコントローラなどを組み合わせて1チップとしたもの〕

iAPX286〔仮想メモリを採用し，論理アドレス空間$2^{30} \fallingdotseq 1$G byte，物理アドレス空間$2^{24} \fallingdotseq 16$M byteを持つ．メモリ保護や特権レベルを有する〕

などがある．

目　次

第4章　8086の命令グループ　(265)

第5章　ソフトウエア開発　(329)

第6章　8086アセンブリ言語のプログラミング例　(343)

第7章　8086マイクロプロセッサ　(357)

第8章　8086の基本デザイン　(389)

第9章 Multibus (449)

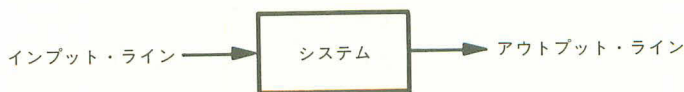
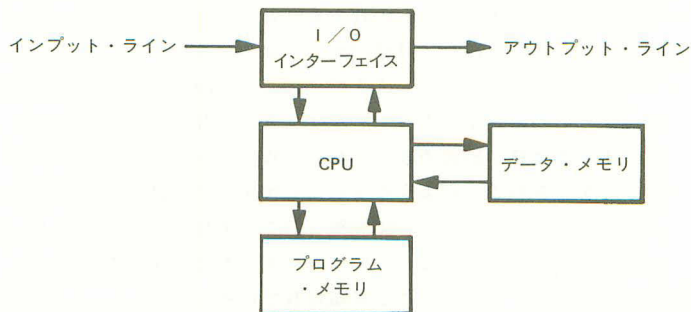
第10章 8086のマルチプロセッサ構成 (463)

付 録 (477)

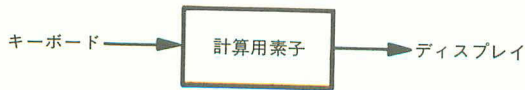
第1章
プログラミング

1.1 アセンブリ言語

マイクロコンピュータのシステムにおけるアセンブリ言語の機能は何か．機械語あるいはより高いレベルのプログラミング言語と，アセンブリ言語がどのように異なっているのか．この章では，アセンブリ言語が果たすいくつかの役割を評価することによって，これらの疑問に答える．

一般的な意味において，すべてのマイクロコンピュータのシステムは次の形式をとる．

ここでインプット・ラインはシステムに情報を供給するために用いられ，アウトプット・ラインはシステムから情報を送るために用いられる．普通，システムは次のものから構成される．

CPU (Central Processing Unit) は，インプット・ラインからI/Oインターフェイスを通してデータを受け取り，ＣＰＵはプログラム・メモリからの命令を実行することによって，このデータを処理する．結果は，アウトプット・ラインを通して出力される．ＣＰＵは，データ・メモリに一時的なデータをたくわえる．

ＣＰＵ，I/Oインターフェイス，そしてメモリは，システムのハードウエア部分であり，プログラム・メモリ中のデータは，システムあるいはプログラムのソフトウエア部分である．8086アセンブリ言語の要素は，処理されてプログラム・メモリに格納されるプログラムを形成するために組み合わされる．したがって，アセンブリ言語は，メモリ中に存在するプログラムを記述するために用いられる．

プログラムの概念を理解するために，次の要素を持つ基本的ＰＯＳ (Point−Of−Sale) 端末を考えてみる．

キーが押されると，計算用素子は，押されたキーを機械が受け入れられるコードへ変換する操作を行なう．コードは処理されるべき数値あるいは実行されるべき計算を表わす．このコードを解釈することによって，計算用素子は必要な処理を実行し，ディスプレイ上に結果を表示する．

計算用素子は，一連のタスクを実行することによって，これらの処理を行なう．たとえば，計算用素子は，キーが押されたかどうかを判断するために，次の一連のタスクを実行する．

1．キーボードのステータス・バイトを読み取る．

2．ステータス・バイトからビット3を取り出す．

（ビット3が0ならば，キーは押されていない．ビット3が1ならば，キーは押されている．）

3．ビット3をテストする．

（ビット3が0ならばステップ1へ戻る．ビット3が1ならばステップ4へ進む．）

4．次のタスクを実行する．これは，キーが押されたことを表わすビットをクリヤするコマンド，あるいはキーボードの動作を無効にするコマンドである．

すべての変換と計算の処理を含む，計算用素子によって実行されるタスクの完全な集合は，アルゴリズムとして知られている．アルゴリズムは，開始点と終了の基準をもつ明確に順序づけられたタスクの系列より成る．アルゴリズムは普通，上述の例，すなわち実行すべきタスクを記述する文章の形式で表わされる．残念ながら，ＣＰＵは“キーボードのステータス・バイトを読み取る”のような文章に応答することはできない．文章で構成されているアルゴリズムから，ＣＰＵによって解釈可能な2進数の命令系列で構成されている形式への翻訳が存在しなければならない．アルゴリズムを実現するために用いられる命令の集合は，オブジェクト・プログラムとして知られている．

ＣＰＵは，２進数すなわち１あるいは０の数字より成る情報の単位を解析することによって，命令を実行する．簡単なＣＰＵは，命令のフェッチと命令実行の２つのサイクルを持っている．命令のフェッチ・サイクルにおいてＣＰＵは，実行されるべき次の命令（情報の単位）を含む位置のアドレスを生成し，メモリがその位置における情報の単位を供給することを要求する．メモリは適切な情報の供給を行なう．次の命令実行サイクルの間に，ＣＰＵはその情報を解析し，適当な動作を実行する．

たとえば，前の例で示したタスクを実行するために，インテルの8086システムにおいて，次のデータが与えられたと仮定する（アドレスと命令は２進数）．

アドレス	命　令
0000	11100100
0001	00001010
0010	00100100
0011	00001000
0100	01110101
0101	11111010

8086が0000から実行を開始したとすると，0000における最初の命令を読み込んで，それを解析する．ＣＰＵは，その命令が入力の命令であり，次の0001がデータの読み込まれるべきデバイスのアドレスを含んでいると判断する．したがって，デバイス・コードは0000 1010となる．もしデバイス・コード00001010のデバイスがキーボードのステータス・バイトを生成するならば、この命令の実行によってキーボードのステータス・バイトが8086のＡＬレジスタに読み込まれる．0000の命令実行後に，次に実行される命令は0010の命令である．0010の命令は，ＡＬレジスタとのＡＮＤを実行するために0011の情報を用いる．これは，前述の例の２番目のタスクにおけるビット３の取り出しである．0100と0101の命令は，ビット３が１か０かを判断し，次いで適当な動作を行なう．

ＣＰＵは１と０を用いて動作する．しかし人は，１と０を用いることにそれほど慣れてはいない．したがって，人とＣＰＵが用いている１と０の間に，中間のステップが設けられている．このステップが，アセンブリ言語である．コンピュータに１と０を直接入力する代わりに，人はアセンブリ言語でプログラムを書く．アセンブリ言語のプログラムは，アセンブラとして知られているプログラムによって適当な１と０とに変換される．アセンブリ言語で書かれたユーザのプログラムは，ソース・プログラムと呼ばれている．

たとえば，前に示したように，１と０より成るプログラムを作成する代わりに，8086アセンブリ・コード（ソース・コード）の以下に示すラインがアセンブラに入力できる．

```
TOP:    IN      AL, 0AH
        AND     AL, 08H
        JNZ     TOP
```

アセンブラはこのコードを，前述の例の１と０（オブジェクト・コード）に変換する．

アセンブリ言語は，アセンブラによってシステムが実行可能な１と０のすべての組合せに変換可能な命令の集合より成る．

たとえば，8086アセンブリ言語の命令

```
AND   AL, 08H
```

は，アセンブラによって次の2バイトのオブジェクト・コードに変換される.

00100100
00001000

このオブジェクト・コードは8086によって，ＡＬレジスタの内容と00001000との間でＡＮＤをとる命令として解釈される.

いくつかの理由から，アセンブリ言語のプログラミングは，２進数コードによるプログラミングよりも効率が良い．第１に，01001000，10100010，あるいは01110000などの命令を書くよりも，ＡＮＤ，ＡＤＤまたはＸＯＲなどのアセンブリ言語の命令を用いてアセンブリ・コードを書くことの方が明らかに容易である．第２に，機械語による命令入力の際に，エラーの可能性は非常に高い．アセンブリ言語の作成の段階では，もしエラーが存在しても，それらは普通，アセンブラによって見つけられる.

1.2 プログラミングの作業

次に，プログラマとマイクロコンピュータ・システムの間の関係について考える．マイクロコンピュータ・システムを働かせるために，プログラマが一般に行なう作業には以下のものがある.

① システムの仕様：システムが取り扱う入力と出力の特性の記述に加えて，システムが提供するすべての機能の一般的な解説が仕様に含まれる.

② 与えられたシステムについての仕様を実現するコンピュータ・プログラムの設計：これには，提案されたシステムが特定の応用に対処できる一連のステップとして，仕様が実現されることが必要とされる.

③ 特定のコンピュータ言語を用いたプログラム設計の実現：この段階には，コーディング，デバッグ，完成の３つの別々の作業が含まれている.

④ 完成したシステムのテスト：多くのテスト・データをシステムに入力する．プログラムのロジックとハードウエアの構成要素を働かせるように，テスト・データは設計される.

⑤ システムの文書作成：適切な文書作成には，全体のシステムがどのように動作するかの記述，システムに対するオペレータの手引き，そしてプログラムの完全な文書化が要求される.

⑥ システムのメインテナンス：新しい要求あるいは新しい装置が必要とされるならば，システムを更新するためのプランが存在しなければならない.

複雑なシステムをプログラミングする場合，上記のリストは常に用いられる．しかし，システムの拡張に関する３つの小節を含む 250ページの仕様と，50ページのオペレータの手引き，さらに厳密なテストの方法が必要とされない場合は限られている．これらの計画は，次のような種類の場合に現われる.

1．シリアルI/Oチャンネルの動作が正常でない．ハードウエア関係者はソフトウエアを指摘し，ソフトウエア関係者はそのようなむさくるしいものが常に正常であること

を信じるのは不可能だと考えている．有望なことにこの解決法は，チャンネルを初期化してシリアルI/Oチャンネルがデータの有効性を示すごとにデータを読んで表示する短いプログラムにある．ハードウエア動作の確証を得ることは比較的容易であり，もしハードウエアが正常ならば，何が働いていないかについての疑問はほとんどなくなる．

2．小数の重要な計算が行なわれる必要がある．幸運にも，FORTARN システムが利用可能である．有望なことに，希望する結果を生じるであろう20ステートメントの FORTRAN プログラムがその解となる．

前述のどちらの場合においても，仕様あるいはプログラムの計画が紙に書かれることは非常に少なく，これらのステップはプログラマの頭の中で行なわれる．これらの場合においては，おそらく文書が作られることはなく，メインテナンスの必要性は疑問となる．しかしこれらの例は，慣例に対して例外であることを覚えていることは非常に賢明である．

作業に関する①から⑥の項目は，多数のプログラマがいるような状況では，非常に有効に用いられる．何人かのプログラマはステップ①と②を専門に行ない，何人かのプログラマはプログラム設計の実現のみを行ない，何人かはプログラムのシステムのテストに時間のほとんどを費やし，他の人は文章やメインテナンスに活動を限定し，さらに他の人はプログラム作業の統括を行なう．このようにしてプログラマは，より高い生産性を可能とする特殊技能を磨いている．しかしながら，普通，アセンブリ言語のプログラマは，前述の作業のすべてを実行することを求められている．

この本では，アセンブリ言語による8086へのアプローチを重視している．そこで，これらの作業すべての一般的な解説を以下に示す．

1.2.1　システムの仕様

マイクロコンピュータの習得を最初に考慮すると，その必要性は次の2種類の解析の中から結果として生じる．

1．直面する特殊な問題が存在している．たとえば，航空宇宙製造業者は，ある大きさと速度の必要条件を満たす搭載用計算システムを必要とするミサイル誘導システムを生産している．

2．新しいマイクロコンピュータ・システムに対する特定の市場が存在する．たとえば，以前は会計のコンピュータ化の余裕のなかった小さな事業所は，マイクロコンピュータをベースとする業務システムの価格が十分に低くなれば購入する．

いずれの場合においても，計画されたシステムが果たすであろう正確な機能を明確にすることは重要である．

第1の場合において，問題の性質は，特定の機能を果たすためにおそらくシステムを制限する．第2の場合においては，仕様のみ考慮すればよい．小規模の業務システムについては，システムにおいてどのような会計の機能が実行されるかを明確に，そして各種のタイプのどれだけ多くの記録が許されるかを，正確に定義する必要がある．そうでなければ，

マイクロコンピュータのシステムは，そのハードウエアが取り扱うことのできる能力以上のタスクを割り当てられることになる．本章の初めのマイクロコンピュータ・システムの簡単なモデルを振り返ると，仕様は次のように定義される．

- システムによって受け取られる入力．
- システムによって実行される計算処理．
- システムによって作り出される出力．

(1) 入 力

マイクロコンピュータ・システムの入力の仕様は，実行されるプログラミングのレベルに大きく依存している．BASICを用いているアプリケーションのプログラマは，ディスク・コントローラに与えられるコマンドのタイプへの関心はあまりありそうにない．むしろ，ディスク上のデータのタイプ，ディスク・ファイルに記録がどのように割り付けられるか，ディスク・ファイルの操作をオペレーティング・システムがどのように圧縮するかなどに関心を持っている．この本自体はアセンブリ言語のプログラミングについて記述しているが，データ・ベースの取り扱いの技法についての適当な解説はこの本の範囲を越えているので，ハードウエア・レベルでの入力と出力について重点的に述べる．

ハードウエア・レベルで，3つのパラメータが入力チャンネルの特性を定めている．それは次のものである．

1．データ・バスの幅：プロセッサ・コントローラ・エラー・システムからは一度に1ビットの入力が到着する．パラレルまたはシリアルのI/Oチャンネルは，一度に8ビットを入力する．フロッピー・ディスク・コントローラは，リクエストによって1024（128 バイト）の情報を伝送する．
2．データ伝送の速度とタイプ（同期あるいは非同期）：リアルタイム・クロックの 200 マイクロ秒ごとに，データが到着する．10ミリ秒ごとに，シリアルI/Oチャンネルは非同期にデータを入力する．コントロール・システムのA/Dコンバータは未定の速度でデータを伝送するが，速くても 500 ミリ秒に1回の速度である．
3．付随のコントロール情報：データが有効になると，フロッピー・ディスクは割り込みを発生する．キーボード・サブシステムは，データが有効ならばステータス・ビットをセットする．新しいデータが有効かどうかを決定するために，システムが入力を読み取り，前のデータと比較することをA/D コンバータは要求する．

これらのパラメータが説明された後で，入力チャンネルがどのように動作するかを述べることは重要である．

入力チャンネルには普通，以下に記す3つのタイプのポートがある．

1．データ・ポート：このポートは，システムの処理部分へ送られるデータを含んでいる．
2．ステータス・ポート：このポートは，いつデータが有効か，このチャンネルにエラーが発生していないかを示す情報と外部に関する情報を含んでいる．
3．コントロール・ポート：このポートは一般に，チャンネルの動作モードを初期化するためと，チャンネルの外部に対する表示方法をコントロールするために用いられる．

この3つのポートは常に存在するとは限らない．ある場合には，データ・ポートだけが存在し，ある場合には，パワー供給時にチャンネルが自動的に初期化され，したがってコントロール・ポートを必要とはしない．

(2) 計算処理

マイクロコンピュータ・システムの計算部分を述べるとき，以下の関連ある3つの主要な領域が存在する．

1．入力部分からの生のデータの処理：これは，データのブロックをその構成部分（たとえば，ディスケットからの1セクタのデータを，ファイル・ヘッダ，ヘッダ・チェックサム，データ，そしてデータ・チェックサム）に分離して，システムによってより容易に利用できるコードへの変換（たとえば，ASCII から2進数へ）の形を取ることができる．

2．システムによって実行される実際のアルゴリズム：実際のアルゴリズムの完全な記述は普通，プログラム設計でなされ，仕様のこの部分ではシステムが実行する主要な機能のすべてを記載しなければならない．

3．出力部分のデータ処理：この処理には，出力装置によって利用可能な形式へのデータの変換（たとえば，2進数データの EBCDIC への変換）が含まれる．

(3) 出 力

マイクロコンピュータ・システムの出力の記述は，入力部分で行なわれたものと非常に類似した解析を必要とする．各出力チャンネルについては，以下の3つのパラメータが存在する．

1．チャンネルによって伝送されるビット数．

2．出力チャンネルにおけるデータ伝送速度．

3．いつトランスミッタがデータを要求するか，あるいはさらにデータを取り扱うことができるかを，システムに知らせる付随のコントロール情報．

これらのパラメータの説明に続いて，チャンネルがどのようにコントロールされるかを述べることが必要である．入力チャンネルと同じく，出力チャンネルは普通，以下に示す重要な3つのポートを持っている．

1．データ・ポート：このポートは外部へ伝送すべきデータを受け取る．

2．ステータス・ポート：このポートは，いつデータがデータ・ポートに伝送されるか，チャンネルでエラーが発生していないかを表わす情報と，外部に関する他の情報を含んでいる．

3．コントロール・ポート：このポートは一般に，チャンネルの動作モードを初期化するためと，チャンネルの外部に対する表現方法をコントロールするために用いられる．

入力チャンネルと同様に，これらのポートのすべてが出力チャンネルをコントロールするために必要ではない．

3つの主要な各部分についての記述の過程で，以下に示す覚えておくべきいくつかの有用な技法が存在する．

1．各部分において，発生する可能性のあるエラーの状態やエラーに対するシステムの

応答の一覧表の作成．

2．各部分において，その部分が取り扱うすべての機能の一覧表の作成．たとえば，すべての入力チャンネル，すべての計算処理機能，さらにすべての出力チャンネルの一覧表の作成．特定の部分が終わったならば，システムに対するすべての可能性が認識されたことを十分確実とするように，その部分を一覧表で相互に照合する．

書かれている最初の仕様が必ずしも最終のものではない．もし手元にある問題があまり簡単なものでないならば，それはほとんど確実に最終のものとはならない．プログラム設計の作業と実施の仕事は，選択されたハードウエア構成が与えられたのでは，確実な機能が実行されるのは不可能なことを示す．この場合には，ハードウエア構成が変えられるように仕様を修正するか，あるいは与えられたハードウエア構成が実行できるように，障害となっている機能を変更する必要がある．

1.2.2 プログラム設計

プログラム設計には，仕様中のことばを解釈して，仕様を実現する方法を示す一連の言語でステップを書くことが含まれている．理想的には，この言語によるステップは，システムが成し遂げるべきことの明白で簡単な記述を与える．この時点で，すべてのシステムに対して簡単な記述が得られることは，直ちに明らかとはならない．たとえば，ＩＢＭのDOS/VSオペレーティング・システムの簡潔な記述を見出すことは期待されない．全体のシステムについてこのことは真実であろうが，理想的立場ではシステムの個々の部分（たとえば，プリンタ・ドライバや複数ワードの減算ルーチン）に対しては，簡単な記述が利用可能であるべきである．非常に大きいシステムに含まれる部分の数を考えると，規模の大きい仕様を多数のずっと小さいモジュールに分割するプログラム設計者の仕事がかすかに頭に浮かぶ．

プログラム設計の仕事に従事するときは，以下の提案を心に留めておかれたい．

1．将来において，より大きい能力を備えるためにプログラムを拡張しなければならない．したがって，プログラムは組み込みによる拡張の機能を備えていなければならない．このような機能には，システム・サブルーチン，データの拡張可能なテーブルやリスト，システムにより多くの機能を付加するための便利でよく説明された方法，そして適度に融通性のあるデータ構造が含まれている．

2．一般的な設計として，与えられた機能を果たすためには１つ以上の方法がある．ある場合には，機械の制限から１つの解法を用いることが強いられる．他の場合には，時間的な制約から別の解法が強いられる．これらの要因，すなわち機械と時間の制限は，実際のコーディングが行なわれる実施作業まで十分知られていないので，設計の段階ではしばしば特殊な問題を解く別の方法を追求することが賢明である．この段階で，他に代わるものを見出すことの利点は２つある．第１に，もし引用した制限が１つの解法を妨げるならば，もう一方は既に利用可能となっており，第２に，その過程でより有効な解法を発見するかもしれない．

3．設計を行なっている間，特定のモジュールが他のモジュールにどのような影響を与

えるかを明白に記すことは非常に重要であり，またそのモジュールに他のモジュールがどのような影響を与えるかを明白に記すことも同じく重要である．モジュール間のこのインターフェイスは，プログラム・モジュールのデバッグと完成時に重要となる．

プログラム設計が完成すれば，仕様を再検討するためにその設計を用いる．仕様と設計との相互の照合によって，設計や仕様における欠陥や手落ちが明らかにされる．正規の基準から設計が再検討されるべきであるという事実を自覚しなければならない．実現と試験の作業が行なわれている間に，プログラム設計の再評価を行なわなければならないような新しい情報が得られる．

1.2.3 プログラムの実現

プログラム実現の作業は，プログラム設計の作業で記述された言語のアルゴリズムを解釈して，特定のマイクロコンピュータ・システムで動作させることから成る．実現の作業に入るには，次の2つの異なる苦労が存在する．

1．コーディング．これは，プログラム設計の作業で作成された言語によるステップを，特別なコンピュータ言語に変換する過程である．
2．デバッグと完成．これは，コーディングの段階でコンピュータ言語に変換されたプログラム設計モジュールからエラーを取り除き，このモジュールを動作するシステムに完成させる過程である．

(1) コーディング

プログラム設計を特別なコンピュータ言語に変換することは，プログラマの簡単な作業の一つとすることができる．もしプログラム設計の機能が正確になされたならば，個々のモジュールは1組の簡潔な文章によって記述される．コーディングでは，次の提案を心に留めておかれたい．

1．可能ならば常に標準のサブルーチンあるいはプログラムを用いるように心がける．サブルーチンは通例，個々にデバッグすることができるという点で非常に有用である．サブルーチンからバグを取り除いた後では，コードのメインをデバッグすることはずっと容易となる．さらに，標準的なサブルーチンは，システムに新しい特徴を付加することを非常に容易なものとしている．
2．コードの文書をできるだけ明確にする．コードの個々のモジュールあるいはセクションについて述べる注釈文に加えて，覚えやすい意味を持つラベルはきわめて重要な価値がある．ラベルの文字数を6あるいはそれ以下に制限しているような，上記の機会を制限するアセンブラがある．しかしほとんどの場合，プログラムとデータ領域の両方のラベルに驚くべき記憶を助ける価値を与える能力が存在しており，十分に生かされるべきである．

各々のプログラム設計のモジュールが適当なコンピュータ言語に変換された後で，コーディングの作業が正確に行なわれていることを確実にし，デバッグの作業を行なう間，可能性のある障害を避けるために，一連の検査が行なわれる．この検査は，ときにはデスク・チェッキングと呼ばれ，コーディング手順の一部分であるが，またデバッグの作業と多

くの要素を共有している．

行なわれる検査には以下のものが含まれる．

1．コードがプログラム設計のモジュールをすべて含んでいることの確認．

2．プログラム設計に含まれているすべての判断がコードに含まれていることの確認．正しく分岐が行なわれることを確実にするための，すべての判断点における論理の検査．

3．各々のプログラム設計のモジュールに，それを正しく動作させるために十分な情報が供給されていることの確認．この検査は，各モジュールについて，このモジュールが他のモジュールに供給することを要求するものが何かを，次の点から決定することによって行なわれる．

- レジスタの内容
- データ構造の内容
- このモジュールによって用いられるI/O装置の状態
- ステータスの設定

4．各モジュールが後続のモジュールに正確な情報を供給することの確認．この検査は，このモジュールが他のモジュールに供給すべきものが何かを，次の点から決定することによって行なわれる．

- レジスタの内容
- データ構造の内容
- 後続のモジュールが用いるI/O装置の状態
- ステータスの設定

5．次の状況を取り扱うためのコードがこのモジュールに記入されていることの確認．

- エラー
- 特殊な場合
- 限界の場合
- 通常の場合

これらの検査が終了した後に，デバッグの手順が始まる．

(2)　デバッグと完成

デバッグと完成の作業は，コードからエラーを除去し，最終的に動作するシステムに，デバッグされたモジュールを完成させることからなる．デバッグの作業中に行なわれる機能は，机上の検査で行なわれる機能によく似ている．デバッグの作業は，システムのハードウエアあるいはシステムのハードウエアのシミュレータで，動作しているコードを調べる間にこの作業が行なわれる点で異なっている．机上の検査では利用できない，デバッグの過程で用いられる一連の道具がある．この道具は普通，常にではないが，デバッガと呼ばれるソフトウエア・モジュールによって提供される．デバッガによって与えられる代表的な特徴には，次のものが含まれる．

- シングル・ステップの機能：この機能は，プログラムのロジックに従う個々の命令の実行を，ユーザに可能とする．

• メモリあるいはレジスタの内容の検討・変更：この機能は，ユーザがメモリやレジスタの内容を調べ，任意に変更することを可能とする．
• ブレイクポイントの機能：この機能は，ある条件に従ってデバッグされているプログラムの実行を，ユーザが中断することを可能にする．代表的なブレイクポイントの条件には，オペランドの参照あるいは命令のフェッチに対する特定のアドレスの参照が含まれる．

プログラムをデバッグするときは，次の提案を心に留めておかれたい．

1．デバッグの過程は，共通に用いられているサブルーチンか，あるいはシステムのサブルーチンのデバッグから始める．もしソフトウエア・システムにおいて最下位レベルのルーチンが適切に機能していることが知られているならば，メインのコードが間違っているか，あるいは誤った方法でシステム・サブルーチンが用いられていると仮定することが可能なように，エラーの原因を発見することは簡単になる．
2．もし可能ならば，個々の仕様の各範囲のデバッグを試みる．個々に仕様の入力部分の各セクション，次いで仕様の計算処理部分の各セクション，次いで仕様の出力部分の各セクションをデバッグするのが適切である．仕様のセクションを別々にデバッグすれば，システムの他の部分からの干渉を受けずに各セクションを調べることができる．理論的には，個々のモジュールのすべてがデバッグされれば，完成の段階ではプログラムのモジュールが相互にインターフェイスするようにデバッグすることが必要となるだけである．

個々のモジュールのすべてがデバッグされれば，完成の段階に入る．この段階では，個々のモジュールがサブシステムに結合され，サブシステムとしてデバッグされる．たとえば，プログラムの入力部分に影響するプログラム設計のモジュールのすべてが結合されてデバッグされる．各サブシステムがデバッグされると，最終的なシステムがデバッグされるまで，他のサブシステムと結合が行なわれる．前に示したように，完成の段階で行なわれなければならない機能は，モジュール（結局はサブシステム）間のインターフェイスが正確に処理されていることの確認だけである．

完成のいずれの段階においても，プログラム設計あるいは仕様作成の作業にまで戻ることが必要となる場合がある．次の例を考える．

1．コーディングの段階で，指定された機能を設けるために必要なコードが，ハードウエアの設計で与えられた以上のメモリを要することが明らかになる．まず，他の方法でより少ないメモリ領域の利用が可能かを決めるために，プログラム設計の作業へ戻る．もしこれでも問題が解決しなければ，どうやら仕様作成の作業に戻り，システムを何かの方法で再構成すべきときである．
2．デバッグの段階で，システムがコントロールすることになっているすべての装置を動作させる試みが行なわれたときに，十分に速いシステムの応答が得られないことが認められる．仕様の入力と出力の部分は完成するまでは一般に別々にデバッグされるので，デバッグの初期の段階ではこの支障は明白にならないかもしれないことに注意．この場合，入力あるいは出力のコードの実行時間が減少しないかを調べるために，コ

ーディングの作業に戻る．もしこれがだめならば，もっと能率の良いアルゴリズムが利用できるかを判断するために，プログラム設計の作業へ戻る．まったくもってこれがだめならば，システムを改革するために仕様作成の段階に戻る．

1.2.4 試　験

試験の作業は，特殊なデータの集合を用いてシステムを十分に動作させて，正しい結果の得られることを確かめることである．この作業は，非常に多くのプログラマが存在する環境ではごく普通のことである．たとえば，自尊心のあるソフトウエア・ハウスがオペレーティング・システムの新しいバージョンを発売する前には，その新しいシステムは十分に吟味されている．自動車製造業者がコンピュータ・システム搭載のバージョンを発売する前には，厳格な試験が実施される．しかし，アセンブリ言語のプログラマが少ない状況では，試験はしばしば見落とされる．試験を無視する主な理由は，それが非常に時間を消費し，その結果非常に経費がかかることにある．その上，試験はよく理解された技術ではない．

有望なことに，試験の作業の重要な部分は実現の作業のデバッグ段階の間に行なうことができる．たとえばデバッグの部分で，境界条件となるデータを用いてモジュールを実行し，それによって判断すべきモジュールの機能を働かせて各モジュールが試験される．例として，データ・ブロックの最初のバイトが30_{16}から39_{16}までの範囲で1つの機能，最初のバイトが41_{16}から46_{16}までの範囲でもう1つの機能，そして最初のバイトがどちらの範囲でもない場合に第3の機能を実行するためのモジュールのコード化を考える．代表的な試験データは，次の値を最初のバイトに持つブロックを適当に含んでいる．

$2F_{16}$
30_{16}
39_{16}
$3A_{16}$
40_{16}
41_{16}
46_{16}
47_{16}
00_{16}
FF_{16}

これらのブロックは，データの異なるタイプを区別するシステムの能力を試験する．

システムに従った試験データを決定するときは，次の提案に留意されたい．

1．入れるべきデータには次の3つの基本的なタイプがある．
   - システムが通常出会うデータの典型的な流れ．
   - 判断を正確に実行するシステムの能力を働かせる一連の境界条件．
   - 論理的データと非論理的データの両方を含むデータのランダムな選択．
2．データは次の速度でシステムに与えられなければならない．

- システムが通常出会う典型的なデータ速度．
- システムが機能するとされている最高のデータ速度．
- データ速度のランダムな選択．

1.2.5 文書作成

文書作成の作業は，システムに関するすべての情報を書き留めることである．システムの文書作成には，次の3つの基本的な要素がある．

1. プログラムに関する文書作成：実際の作業の解説のところで示したように，モジュールごとに，コードがどのように働き，ある場合にはなぜコードがそのように動作するかを説明することは非常に重要である．この種類の文書作成は，コードの新しい読者にもコードと容易になじむことを可能とする．さらに，コードを変更する必要が生じても，変更がどのように行なわれるかの合理的で精通した判断が，すぐれた文書によって可能となる．プログラムに関する文書は，ずっと以前に書かれたプログラムを調べたり，最初にコード化を行なった人の記憶を新たにする手助けとなる．
2. システムの手引き：システムの手引きには，プログラム設計の記述，プログラム変更方法の記述．システムが外部で何が起きることを予期しているか，すなわち，何が入力ラインを駆動し，何が出力ラインでデータを受け取るかの簡潔な要約が含まれるべきである．有望なことに，システムの手引きは，以前の作業の間に少なくとも文書の形で主要な要素が詳細に書かれているはずなので，かなり簡単にまとめられる．
3. ユーザの手引き：これは文書の中で最も重要な部分である．もしコードに関する文書がよく整っていて，さらに最も良いシステムの手引きが書かれていたならば，他のプログラマはプログラムを変更あるいは改良することが可能となる．しかし，もしユーザの手引きがなければ，だれもプログラムを使用することはできず，この場合，コードを変更あるいは改良しようとする人はいなくなり，すべての努力が無駄となる．システムとインターフェイスするプログラムを外部のユーザが書くようなシステムでは，ユーザの手引きは特に重要である．この場合，システムの改訂あるいは付加は，典型的にはユーザの手引きの更新を通して，ユーザに知らせることが最も重要である．

1.2.6 メインテナンス

メインテナンスの作業は，新しい設備あるいは新しい処理要求に適応するためのプログラム変更より成り，本質的には変化する環境でプログラムの機能を保つことにある．メインテナンスの作業は，環境の変化に応じて，簡単にも複雑にもなる．簡単な作業の例には次の事項が含まれる．

1. 装備の時代遅れの部品に代わって，システムにハードウエアの新しい部品が取り付けられる．I/Oインターフェイスは古いハードウエアと非常に似ていて，実際，変更はいくつかのステータス・ラインの入れ替えだけを含んでいる．この場合，メインテナンスの作業には，プログラムの数行のコードの変更，新しいハードウエアを用いたこのコードのデバッグ，さらに文書への適当な付加事項の記録が含まれる．

2．いつか後でより複雑なシステムによって処理される可能性のあるディスケット・ファイルに，システムが出力を行なっている．より複雑なシステムの要求によって新しいオペレーティング・システムが作られ，重要なある目的のために，ディスケット・ファイルで以前使われていなかった2バイトが用いられる．この可能性はプログラム設計の段階で考慮されるので，この場合のメインテナンスの作業は，仕様決定作業の計算処理の部分と関連したプログラムの設計，実現，さらに文書作成の作業に小さい変更を要するだけである．

より複雑な作業には次のものが含まれる．

1．ハードウエアの新しい部品がシステムに付加される．前の例と比較して，この装置は他のシステムの装置と少しも類似性がなく，実際，インタラプト構成，システムのタイミング，さらにシステムの処理能力について新しい要求が出される．この場合，メインテナンスの作業は確認されているプログラミングの作業の各々において広範囲にわたる労力を必要とする．

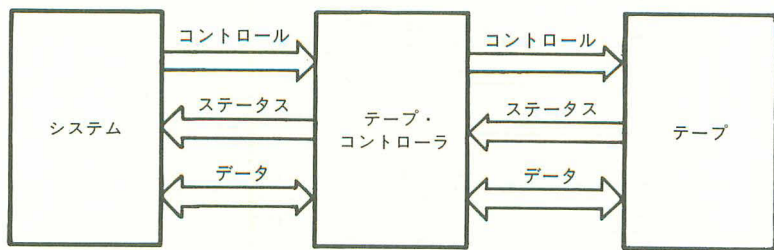
2．販売部門で，マイクロプロセッサに基づくシステムに80メガバイト・ディスク・ドライブの実集合体を付加することを決定する．この場合のメインテナンスの作業には，仕様から文書作成までのプログラミングの作業のすべてが多分含まれるだろうし，あるいは想像できるように能力集団から適当な販売人員の順序づけられた配置換えを含むことになる．

比較的容易にメインテナンスの作業を行なう機能は，プログラム設計と実現の段階で払われた配慮に直接に比例する．プログラム設計の段階で特徴を付加する容易な方法が残されていないか，あるいは一般的システムのモジュールが与えられていなければ，どんなによくみてもシステムへの付加は困難であることが多分証明される．プログラム実現の過程で，プログラムの方法と理由についての合理的な文書が供給されていなければ，コードにプログラム設計の新しい要素を導入することはまったく遠回りなものとなる．

第2章
プログラム例

2.1 ソート・プログラム

ソート・プログラムのモジュールの仕様決定とプログラム設計の作業について考える．このプログラムは，テープ・ドライブのファイルからデータ・レコードを読み取り，レコードから分類キーを取り出し，テープにデータ・ファイルに続いてキー・ファイルを書き込む．このプログラムの実際のコードは，第6章に示す．

この例では，非常に簡単なI/Oインターフェイスを仮定する．I/Oインターフェイスに対する一般的なブロック図を次に示す．

テープ・コントローラは，テープに対して128バイト・ブロックのデータを伝送する．コントローラは，テープに書き込まれるブロックに，パリティ・ビット（9トラックのテープに対して）とチェックサムを付加する．コントローラは，読み込み動作が行なわれるときにこのエラー検出の情報を処理し，それに応じてエラー・ビットを設定する．

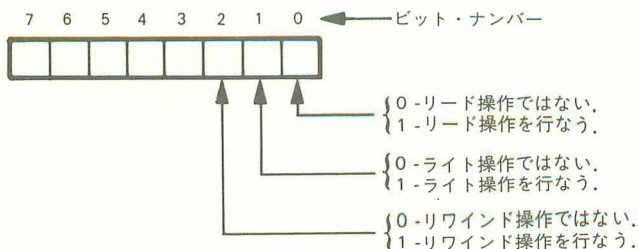
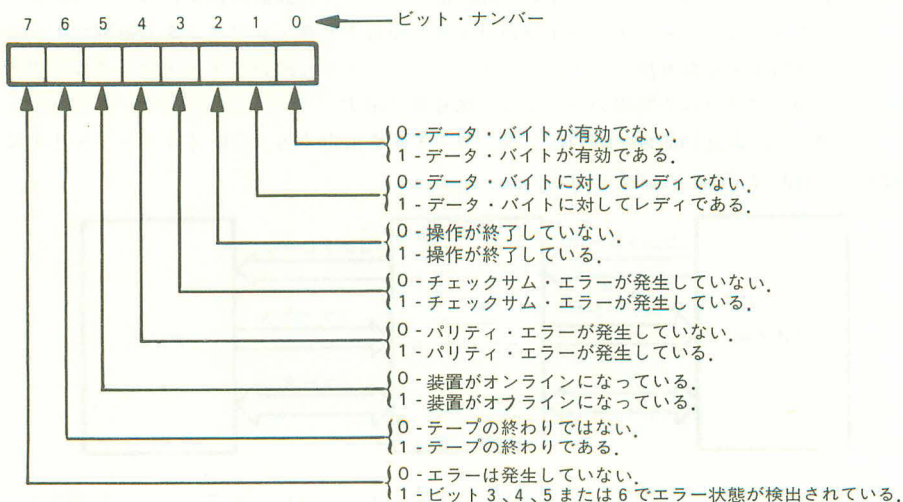
テープとの間の伝送は，次のように実行される．

1．システムは，読み込みあるいは書き込みの動作を要求する．

2．システムは，コントローラが1バイトの伝送の用意ができるのを待つ．

3．システムは，コントローラのデータ・ポートとの間で1バイトを伝送する．

4．もし128バイトが伝送されれば，テープ・コントローラはブロック全体が伝送されたことを表わすフラグをセットする．伝送が完了していなければ，システムはステップ2へ戻る．

テープ・コントローラは，非常に簡単なコマンドの構成で動作する．次に示すコマンド・バイトは，テープ・コントローラによって動作を起こさせるために，テープ・コントローラのコマンド・ポートに送られる．

テープ・コントローラへのコマンドの送出の後，システムはコントローラからステータス・バイトを読み込む．このバイトは次の形式を持っている．

システムがリード動作あるいはライト動作のコマンドを発行すれば，適当なビット（リード動作に対してはビット0，ライト動作に対してはビット1）がサンプルされる．テープ・コントローラにおいてデータの送受が可能ならば，システムはテープ・コントローラのデータ・ポートに対して読み込みあるいは書き込み動作を行なう．128回の読み込みまたは書き込みの動作の後に，ビット2が1となって動作の完了を知らせる．

システムがリワインド動作のコマンドを発行すれば，ステータス・ポートのビット2はリワインド動作の完了を表わす．

2.1.1　入　力

テープ・コントローラの特性についての前の記述が与えられれば，入力のパラメータは以下のように指定される．

1．データ・パスの幅．テープ・コントローラからデータは一度に1バイト到着する．システムからの各リード・コマンドは，テープから128バイトのブロックの読み取りを可能とする．
2．データ伝送の速度．この例では，データは同期的に到着する．データ・バイト有効のビットがハイになると，データはCPUの最大速度で入力される．
3．付随のコントロール情報．この例では，テープ・コントローラはシステムに割り込みを用いていない．システムは，データが有効かどうかを決定するために，テープ・コントローラのステータス・ポートを読み取る．

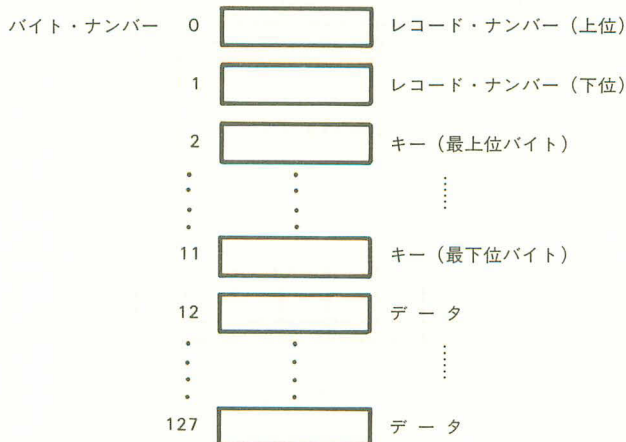
2.1.2　計算処理

仕様決定のこの部分では，次の要求が考慮される．

- テープから読み出されるデータ・レコードのフォーマット．
- キーがソートされる方法．
- テープに書き込まれるデータ・レコードのフォーマット．

2.1.3　入力レコードのフォーマット

テープから読み出された各データ・レコードは，128バイトから成る．テープから読み出された各データ・レコードには，次の関連した3つのフィールドが存在する．

1．レコード・ナンバー．これは，一意的にレコードを識別する2バイトのフィールドである．レコード・ナンバーは 0000_{16}－$FFFF_{16}$ の範囲にある．レコード・ナンバー $FFFF_{16}$ はＥＯＦ(End—Of—File) レコードを指定する．

2．キー．これは10バイトのフィールドである．このフィールドは，レコードの詳細を示すデータを含み，特定のレコードに対して一意的である必要はない．この例では，この10バイトは個人の姓を表わすと仮定している．

3．データ．レコードの残りの116バイトはデータを含む．

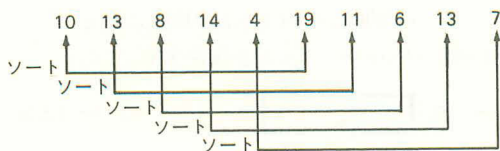
これらの3つのフィールドは，すべてのレコードに対して17ページの図のように構成されている．便宜上，データ・レコードのサイズはテープから読み出されるブロックのサイズに等しいことに注意．

2.1.4 ソートの方法

用いられているソートの方法は，漸減増加ソート，あるいはShellソートである．これは一般に用いられているソートのアルゴリズムであり，"Sorting and Searching," by D. W. Knuthに詳細に述べられている．用いられている照合の順序は，ASCII照合の順序である．キーは上昇順に分類される．

漸減増加ソートの基本的原理は，全体のリストが直接挿入を用いてソートされる最終のパスとなるまで，直接挿入法を用いて順次より大きいサブリストをソートすることである．このソートの利点は，サブリストがソートされるにつれて全体のリストがより整理されたものとなることにある．したがって，最後のパスで全体のリストがソートされるとき，必要な交換が少なく，しかも実行時間が減少する．例として，10個の要素より成る次のリストを考える．

10 13 8 14 4 19 11 6 13 7

最初のソートのパスでは，次のようにリストがソートされる．

10 11 6 13 4 19 13 8 14 7

2番目のパスでは，次のようにリストがソートされる．

4 7 6 8 10 11 13 13 14 19

そして最後のパスでは全体のリストがソートされる．

基本的なアルゴリズムを次に示す．

与えられるのは，Nレコードである．この場合，レコードは12バイトの長さで，レコード・ナンバーとキー・フィールドから成る．このアルゴリズムには，関連のある2つの変数が存在する．

Increment：漸減増加ソートでは，サブリスト中の要素数を決定するのに役立つ1組のIncrementが選ばれる．この場合，Incrementは次のようになる．

N/2，N/4，……，1

N/2，次にN/4，そして最後に1（全体のリストがソートされる最後のパスで）を値として含む変数をIncrementと呼ぶ．

Subsort counter：Incrementの各値に対して，すなわちソートの各パスに対して，この変数は（N−Increment）からNまでカウントする．これは各パスにおいて実行されるソートの数を決定する．

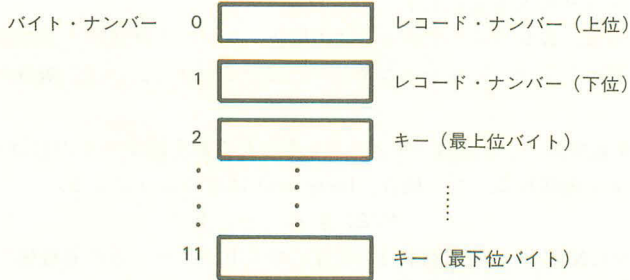
アルゴリズムは次のように処理を行なう．

1．Increment＝N　とする．
Increment＝0となるまで，ステップ2から12までを行なう．
2．Increment＝Increment / 2
直接挿入ソートを用いて，各サブリストをソートする．
3．Subsort counter＝N−Increment
Subsort counter＝N＋1となるまで，ステップ4から12までを行なう．
4．Subsort counter＝Subsort counter＋1
5．Keytemp＝Key(Subsort counter)
6．Recordtemp＝Record(Subsort counter)
7．Index＝Subsort counter−Increment
8．KeytempとKey(Index)を比較．
Keytemp≧Key(Index)ならばステップ12へ，そうでなければステップ9へ行く．
9．Record(Index＋Increment)＝Record(Index)
10．Index＝Index−Increment
11．Index＞0ならばステップ8へ，そうでなければステップ12へ行く．
12．Record(Index＋Increment)＝Recordtemp

2.1.5 出力レコードのフォーマット

テープに書き込まれる各データ・レコードは12バイトから成る．テープに書き込まれる各データ・レコードには関連する次の2つのフィールドがある．

1．レコード・ナンバー．これは，入力レコードのフォーマットにおけるレコード・ナンバーと同一の2バイトのフィールドである．
2．キー．これは，入力レコードのフォーマットにおけるキー・フィールドと同一の10バイトのフィールドである．

これらのレコードは次のように構成されている.

テープのブロックに含まれるバイト数の128は，出力レコード中のバイト数である12の倍数ではないことに注意．したがって，出力レコードをテープのブロックに詰めるために，何かのアルゴリズムが用いられなければならない．このアルゴリズムは，プログラム設計の項で後述する.

2.1.6 出　力

テープ・コントローラの特性についての前述の記述が与えられれば，出力のパラメータは以下のように指定される.

1．データ・パスの幅．データは，テープ・コントローラに一度に1バイト送られる．システムからの各書き込み命令によって，テープに128バイトのブロックが書き込まれる.
2．データ伝送の速度．この例では，データは同期的に送られる．データ・バイトのレディのビットがハイになれば，データはCPUの最大速度でコントローラに送られる.
3．付随のコントロール情報．この例では，テープ・コントローラは，データに対するレディを知らせるために，システムに割り込みを用いていない．システムは，テープ・コントローラのデータに対するレディを判断するために，コントローラのステータス・ポートを読み出す.

2.1.7 エラー処理

この例では，関連があるのはテープ・エラーだけである．このエラーは，テープに対する読み込みや書き込みのサブルーチンによって処理される．この処理は第6章で論じる.

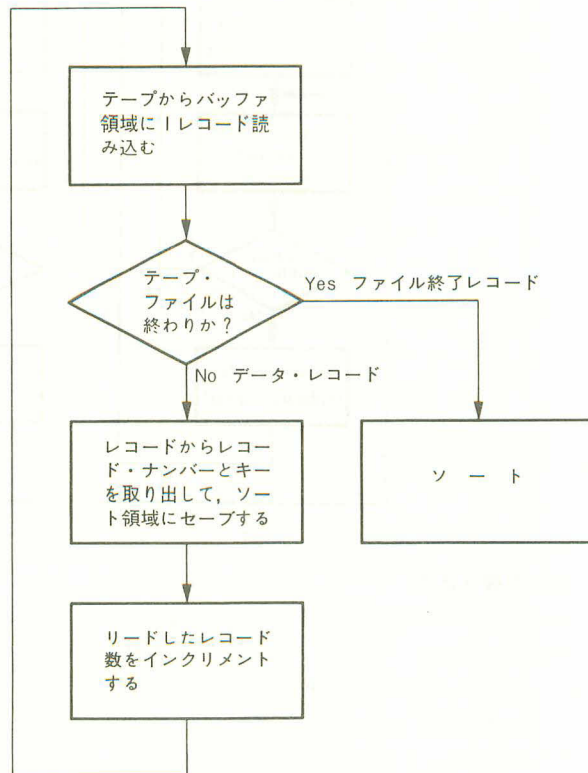
2.1.8 プログラム設計

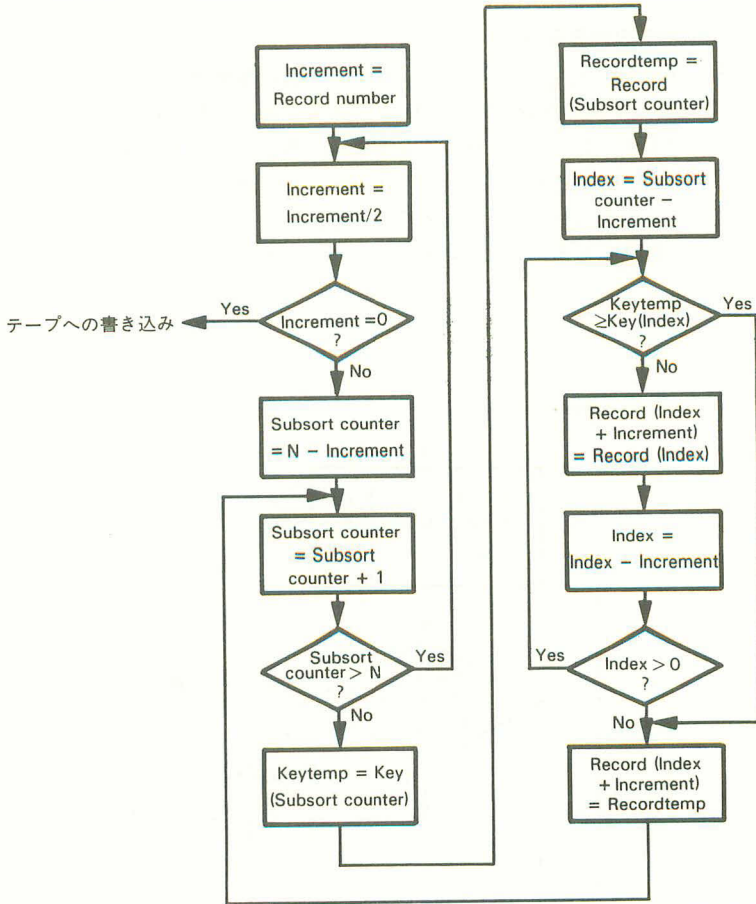
プログラムが行なう作業を検討すると，プログラムを構成する次の3つの主要な機能の存在がわかる.

- テープからレコードを読み出し，各レコードからキーを取り出す.
- キーをソートする.
- ソートされたキーを再びテープに書き込む.

前述のモジュールのどれもそれほど複雑ではないので，それらを説明するためにフローチャートを用いる．

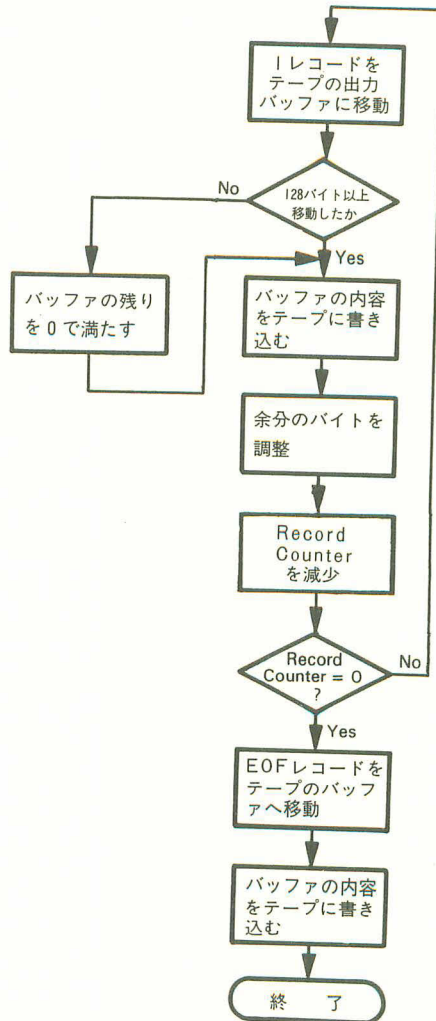
(1)　テープからの読み込み

テープを読み込むモジュールには，ただ1つの判断の個所が含まれている．モジュールは，テープからレコードを読み込みながら，そのレコードがEOFレコード（レコード・ナンバー=$FFFF_{16}$）かどうかを確かめるために，各レコードを調べる．EOFレコードが検出されれば，コントロールはソートのモジュールに渡され，そうでなければ，レコードからレコード・ナンバーとキーが取り出されて，ソートのモジュールによって処理される一時的領域に格納される．それから次のレコードがテープから読み出される．

(2)　ソート

仕様で与えられたソートのアルゴリズムを，このモジュールは実行する．

(3) テープへの書き込み

テープへキー・ファイルを書き込むモジュールは，テープから読み込みを行なうモジュールのように簡単ではない．モジュールには2つの判断の個所が存在する．第1の判断では，テープ・ブロックを満たさなければならない．各12バイトのレコードに対して128バイトのブロックを書き込むことは，テープ・スペースの点からあまり効率が良くないので，128あるいはそれ以上のバイトが格納されるまで，レコードはバッファ内に編成される．

128バイトが格納されると，判断の時点でバッファはテープに一度に移される．第2の判断の個所には，レコード数の減少が伴う．出力レコードのすべてが移されると，バッファにＥＯＦレコード（レコード・ナンバー＝FFFF_{16}）が付加されてテープに書き込まれる．

1レコードを
テープの出力
バッファに移動
128バイト以上
移動したか
No
Yes
バッファの残り
を0で満たす
バッファの内容
をテープに書き
込む
余分のバイトを
調整
Record
Counter
を減少
Record
Counter = 0
?
No
Yes
EOFレコードを
テープのバッフ
ァへ移動
バッファの内容
をテープに書き
込む
終　了

第3章
8086アセンブリ言語の命令セット

8086は，インテルの最初の16ビット・マイクロプロセッサである．1978年に紹介されたとき，従前のマイクロプロセッサと比較して非常に強力なものであった．

8086アセンブリ言語の命令セットは8080Ａと上位方向に互換性がある．ただしソース・プログラム・レベルに限る．すなわち，すべての8080Ａアセンブリ言語の命令は，8086アセンブリ言語の１つあるいは複数の命令に変換できる．だれかが8086アセンブリ言語の命令を，一度に１つずつ，8080Ａアセンブリ言語の１つあるいはそれ以上の命令に変換することを試みる理由はないが，もし行なったならば，すぐさま絶望的にもメモリ配置と特殊な変換規則の矛盾で混乱することになる．これが，8086と8080Ａのアセンブリ言語の命令セットが，“上位方向”に互換性があるという理由である．

8086と8080Ａのアセンブリ言語の命令セットは，オブジェクト・コード・レベルでは互換性はない．このことは，ＲＯＭに記憶された8080Ａのプログラムは，8086のシステムでは使用できないことを意味する。

8085と8080Ａのアセンブリ言語の命令セットは，8085のＲＩＭとＳＩＭの命令を除いて，同一である．8085のＲＩＭとＳＩＭの命令は8086の命令に変換できない．これはＲＩＭとＳＩＭの命令が，8086には存在しない8085のシリアルI/Oロジックを用いていることによる．ＲＩＭとＳＩＭの命令を除いて，8085と8080Ａのアセンブリ言語の命令セットは同一である．したがって，ＲＩＭとＳＩＭの命令は別として，8086アセンブリ言語の命令セットはまた，8085アセンブリ言語の命令セットと上位方向に互換性があることになる．

8085と8080Ａのアセンブリ言語の命令セットは，8085のＲＩＭとＳＩＭの命令を除いて，オブジェクト・コードで互換性がある．すなわち，ＲＯＭに存在するプログラムは，どちらのマイクロプロセッサでも用いることができる．

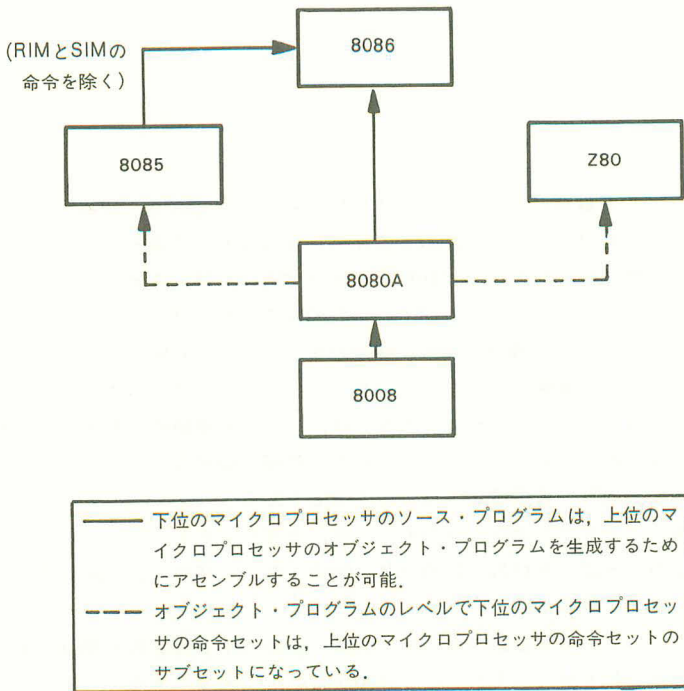
8080Ａアセンブリ言語の命令セットは，Ｚ80アセンブリ言語の命令セットのサブセットとなっている．すなわち，Ｚ80は8080Ａのオブジェクト・プログラムを実行できるが，その逆は成り立たない．Ｚ80の全命令セットが用いられているときは，8080ＡはＺ80のプロ

グラムを実行できない．8086アセンブリ言語の命令セットは，Ｚ80アセンブリ言語の命令セットと上位方向に互換性はない．

歴史的な注釈として，8080Ａに先行した8008マイクロプロセッサもまたソース・プログラム・レベルでのみ互換性があったことは述べる価値がある．すなわち，すべての8008アセンブリ言語の命令に対して，8080Ａのアセンブリ言語の命令が存在するが，２つのマイクロプロセッサのオブジェクト・コード・セットは同じではない．

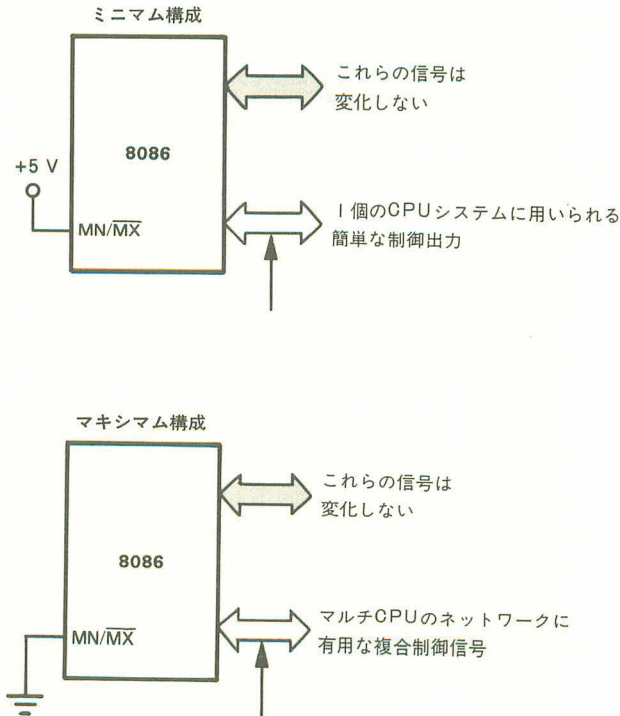
以上に述べた種々の命令セットの互換性は，次のように表わせる．

—— 下位のマイクロプロセッサのソース・プログラムは，上位のマイクロプロセッサのオブジェクト・プログラムを生成するためにアセンブルすることが可能．

--- オブジェクト・プログラムのレベルで下位のマイクロプロセッサの命令セットは，上位のマイクロプロセッサの命令セットのサブセットになっている．

8086のハードウエア設計には，以下のような非常に興味深い革新が見られる．

1．8086のＣＰＵのロジックは，エグゼキューション・ユニット（ＥＵ）とバス・インターフェイス・ユニット（ＢＩＵ）に分かれている．この両者は非同期に動作する．ＢＩＵは，外部バスとのインターフェイスすべてを処理し，外部メモリとI/Oのアドレスの生成を行ない，６バイトの命令オブジェクト・コードのキューを持つ．ＥＵがメモリあるいはI/Oの素子にアクセスする必要が生じると，ＢＩＵに対してバス・アクセスの要求を出す．ＢＩＵが現在動作中でなければ，ＥＵからのバス・アクセスの

要求を受け付ける．EUからの有効な保留されたバス・アクセスの要求がなければ，BIUは6バイトの命令オブジェクト・コードのキューを満たすために，命令フェッチのマシン・サイクルを実行する．CPUはキューの前から命令オブジェクト・コードを得る．したがって，命令フェッチの時間は大きく削除される．

2．8086は，簡単な1個のCPUシステムから複数のCPUのネットワークまで，広い範囲のマイクロコンピュータ・システム構成で動作するように設計されている．この大きい融通性をサポートするために、8086ピンの出力のいくつかは信号の切換えを行なう。これは以下のように説明できる．

MN/$\overline{\text{MX}}$のレベルに基づいて，同一のピンがこれら2つの信号の組を出力する．この大規模な信号の再配置は，マイクロプロセッサ産業にとって最初は非常に空想的であり革新的なものであった．

3．8086は，マルチCPU構成でバス・アクセスのプライオリティを処理するためのロジックが組み込まれている（これは新しい概念ではなく，National SemiconductorのSC/MPでは何年も用いられている）．

4．マルチＣＰＵ構成では，各々の8086ＣＰＵはそれ自身のローカル・メモリを持つことが可能であり，同時に共通メモリを共有する．共通メモリはすべてのＣＰＵあるいは特定のＣＰＵによって共有される．

5．8086は，ミニコンピュータの領域であるプログラムが強調される応用分野においても，有効に競えるように設計されている．外部メモリの１メガバイトまで，直接にアドレス指定ができる．すべてのメモリのアドレス指定はベース相対となっている．このメモリ・アドレス指定方法は当然，リロケイト（再配置）可能なオブジェクト・プログラムを生成する（リロケイト可能なオブジェクト・プログラムは，１つのメモリ・アドレス域から他へ移動して修正なしに再び実行することができる）．また，8086はスタック相対のアドレス指定を用いているので，リエントラント・プログラムが容易に書ける（リエントラント・プログラムは，実行の途中で中断して再び実行することができる．たとえば，自分自身を呼び出すサブルーチンはリエントラントであり，外部割り込みによって実行の途中で中断されて，インタラプト・サービス・ルーチン内で再び実行できるプログラムもまたリエントラントである）．

6．8086は，後続の命令のオブジェクト・コードの解釈を修飾するプレフィックス命令を用いている．

8086は，8080Ａのように，実際は複数チップのマイクロプロセッサ構成の一要素である．8086マイクロプロセッサ自身に加えて，8284クロック・ジェネレータ／ドライバが必要である．他のロジックを用いて必要なクロック信号を作ることもできるが，これは実用的でも経済的でもない．

8086マイクロプロセッサ構成で必要な３番目の素子は，8288バス・コントローラである．8080Ａとシステム・バスの間には，通常8228システム・バス・コントローラを用いるように，8086とシステム・バス（複数の場合もある）の間には普通8288バス・コントローラを用いる．しかし8086の場合，シングル・バス構成では，何の不利益も受けずに，8288バス・コントローラを省略することができる．

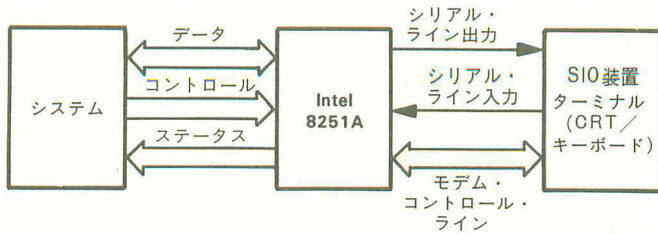
６，７，８，と９章では，基本的な8086のハードウエア，シングルＣＰＵ構成，Multibus*，さらにマルチＣＰＵ構成について論じる．

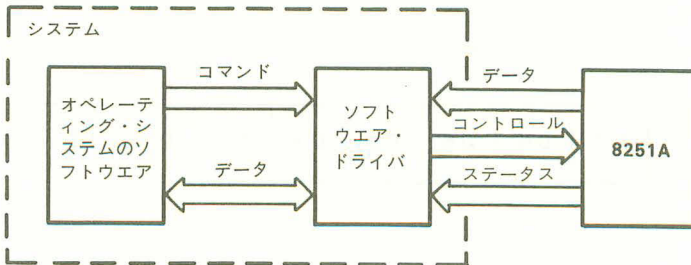
3.1 I/Oドライバ

次に，システムとシリアルの入出力チャンネルとのインターフェイスを行なうプログラム・モジュールの詳細を述べる．

以下にシリアルな入出力チャンネルの一般的なブロック図を示す．

---

* Multibus は Intel Corporation の登録商標である．

この例では，シリアル入出力（ＳＩＯ）のチャンネルは，インテルの8251Ａプログラマブル・コミュニケーション・インターフェイスである．8251Ａは通信端末に接続されていると仮定する．通信端末は，システムから伝送されたデータを表示するＣＲＴを有している．キーボードからのデータは，チャンネルによってシステムへ伝送される．入力あるいは出力に端末で，データにバッファは用いられない．データの送信と受信は非同期に行なわれる．

ドライバとも呼ばれるプログラム・モジュールは，オペレーティング・システムのソフトウエアと8251Ａとを結合している．これは以下のように説明できる．

オペレーティング・システムのソフトウエアは，I/Oドライバのプログラム・モジュールにコマンドやデータを送る．これは種々の方法で処理される．それを以下に示す。

1．コマンドやデータをレジスタに設定する．たとえば，コマンド保持に１つのレジスタを割り当て，データは他のレジスタを通して受け渡す．
2．タスク・ブロックを用いる．タスク・ブロックには，コマンドとデータあるいはコマンドとデータのポインタを含むことができる．タスク・ブロックは，固有のメモリ位置に配置するか，あるいはレジスタの１つで位置を表わすことができる．
3．スタックを用いる．システム・ソフトウエアは，タスク・ブロックと同等のもの（すなわち，コマンドとデータあるいはポインタ）をスタックにプッシュすることができる．

上記方法からの選択は，一般にプロセッサに依存して決定される．現在の議論はプロセッサに依存していないので，パラメータ受け渡しの方法を選択するための原理は，後の章に譲る．

3.1.1 入　力

この例で用いられているＳＩＯ素子はインテルの8251Ａである．この素子は以下の入力仕様を必要とする．

1．データ・パスの幅．8251Ａでは，５，６，７，あるいは８ビットのキャラクタが許されている．この例では，コマンドとステータスに対して，素子で８ビットのデータ・パスを必要とする．異なる大きさのデータ・パスを用いる将来のシステムを可能とするために，プログラム設計ではデータ・パスの大きさを指定することができる．

2．データ転送速度．この例では，データは非同期に転送される．最大データ転送速度だけが指定できる．この例では，最大データ転送速度として，9600ボー（baud）が指定されている．

3．ハンドシェイクのプロトコル．この例では，ＳＩＯチャンネルはマイクロプロセッサにインタラプトを用いているのではなく，マイクロプロセッサが，データが有効かどうかを決めるために，チャンネルを調べる．

次に，I/Oドライバは実際のI/Oチャンネルの動作を考慮する必要がある．データの転送に加えて，8251Ａの場合には，I/Oチャンネルにコントロール情報を送信し，それからステータスを受信しなければならない．

データ・ポートは次のような構成となる．

D7	D6	D5	D4	D3	D2	D1	D0

適当なステータス・ビットが１に設定されると，このI/Oポートのデータが有効であることが知られている．8251Ａの場合，ステータス・ポートのRxRDY（レシーバ・レディ）ビットが１になる必要がある．8251Ａは，コントロール・ポートに情報を書き込むことによって，既知の状態に初期化される．8251Ａの初期化には，少なくとも２バイトのコントロール情報を必要とする．コントロール情報は次の順序で送られる．

1．モード・セレクト・バイト
2．Sync キャラクタ１（同期モードのみ）
3．Sync キャラクタ２（同期モードのみ）
4．コマンド・セレクト・バイト

この例では，8251Ａは非同期モードで動作させるので，以下の初期化の２バイトが必要となる．

1．モード・セレクト・バイト
2．コマンド・セレクト・バイト

モード・セレクト・バイトの形式を次に示す.

コマンド・セレクト・バイトの形式を次に示す.

D7 D6 D5 D4 D3 D2 D1 D0 ← ビット・ナンバー

EH	IR	RTS	ER	SBRK	RxE	DTR	TxEN

TxEN: トランスミット・イネーブル
1＝イネーブル
0＝ディスエーブル

DTR: データ・ターミナル・レディ
“ハイ”にすることによって$\overline{\text{DTR}}$出力を0にする.

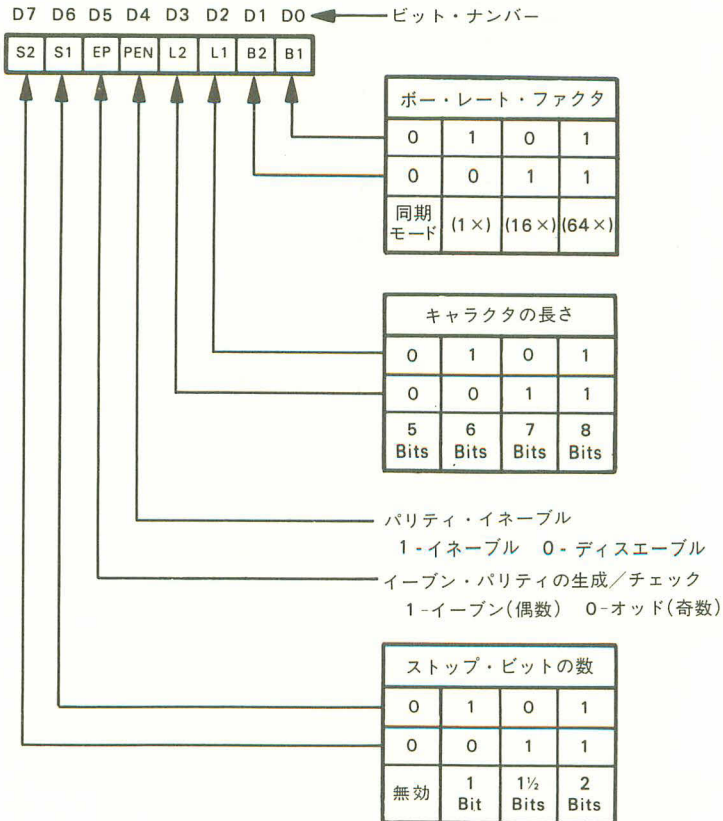
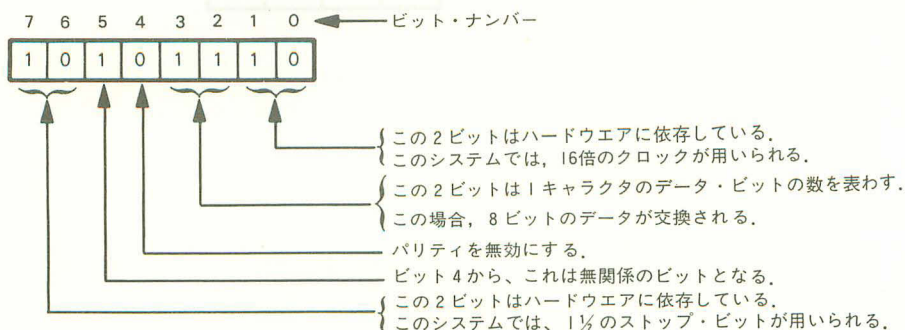
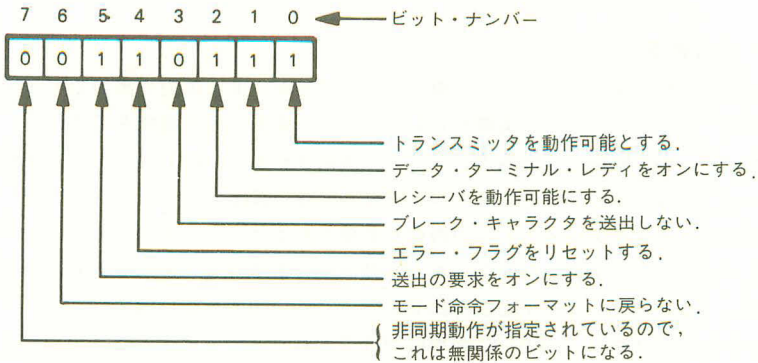
RxE: レシーブ・イネーブル
1＝イネーブル
0＝ディスエーブル

SBRK: ブレイク・キャラクタ送出
1＝TxDを“ロー”にする.
0＝通常動作

ER: エラー・リセット
1＝エラー・フラグPE, OE, FEをリセットする.

RTS: 送出の要求
“ハイ”にすることによって$\overline{\text{RTS}}$出力を0にする.

IR: 内部リセット
“ハイ”にすることによって8251Aはモード命令フォーマットに戻る.

EH: エンター・ハント・モード
1＝同期キャラクタの探索を可能にする.
(非同期モードでは無効)

前述の仕様から, モード・セレクト・バイトは次のようになる.

最初のコマンド・セレクト・バイトは次のようになる.

8251Aプログラミングのその他の特徴は，第6章で解説する.

ステータス・ポートは，8251Aの状態と，それに接続されている装置の状態についての情報を与える.

ステータス・ポートから1バイトが読み出されると，以下の情報がシステムに転送される.

D7	D6	D5	D4	D3	D2	D1	D0
DSR	SYN-DET	FE	OE	PE	Tx-EMPTY	RxRDY	TxRDY

- D0 (TxRDY): このビットはデータ・キャラクタに対してトランスミッタがレディであることを表わす．もしこのビットが1ならば，データは8251Aへ送られる．もしこのビットが0ならば，データは8251Aへ送られない．
- D1 (RxRDY): このビットはシステムに転送可能なデータ・キャラクタがレシーバのバッファにあることを示している.
- D2 (Tx-EMPTY): このビットは送信すべきキャラクタが8251Aにないことを示している．もしこのビットが1ならば、トランスミッタは空である．もしこのビットが0ならば、トランスミッタはデータを送出している.
- D3 (PE): パリティ・エラー．ＰＥフラグはパリティ・エラーが検出されたときにセットされる．これはコマンド命令のＥＲビットによってリセットされる．ＰＥは8251Aの動作を妨げることはない.
- D4 (OE): オーバーラン・エラー．ＯＥフラグは次のキャラクタが有効となる前にＣＰＵがキャラクタを読み込まないときにセットされる．これはコマンド命令のＥＲビットによってリセットされる．ＯＥは8251Aの動作を妨げることはない．しかし，以前のキャラクタは失なわれる.
- D5 (FE): フレイミング・エラー（非同期のみ)．ＦＥフラグはキャラクタの最後に有効なストップ・ビットが検出されないときにセットされる．これはコマンド命令のＥＲビットによってリセットされる．ＦＥは8251Aの動作を妨げることはない.
- D6 (SYN-DET): このビットは8251AがSYNCキャラクタを検出したかどうかを表わす．このビットは8251Aが同期モードで動作するように初期化されている場合だけ意味がある．もしこのビットが1ならば，SYNC　キャラクタが検出されている．もしこのビットが0ならば，　SYNC　キャラクタは検出されていない.
- D7 (DSR): このビットは8251Aの$\overline{\text{DSR}}$ピンの状態を表わす．もしこのビットが1ならば，$\overline{\text{DSR}}$はハイで，したがってデータ・セットはレディでない．もしこのビットが0ならば，$\overline{\text{DSR}}$はローで，データ・セットがレディであることを表わす．これはモデム・コントロール・ラインである.

3.1.2 計算処理

どのような種類の機能をドライバは備えているべきか．また入力や出力でデータはどのように変換されるのか．以下に実際のI/Oドライバが備えている機能を示す．

1．チャンネルの初期化：電源がシステムに供給されると，8251Aは不明の状態で立ち上がる．I/Oドライバはチャンネルを既知の状態に設定する．
2．1個のキャラクタの入力：この機能が要求されると，ドライバは，ステータス・ポートから読み出しを行ない，データが有効になるまで待つ．データが有効になれば，ドライバはデータ・ポートを読み出してシステムに情報を受け渡す．
3．1個のキャラクタの出力：この機能が要求されると，システムは，出力されるべきキャラクタあるいはそのキャラクタのポインタを，ドライバに受け渡さなければならない．ドライバは，ステータス・ポートの読み出しを行ない，トランスミッタが利用可能となるまで待つ．トランスミッタが利用可能となれば，ドライバは指定されたキャラクタをデータ・ポートに転送する．
4．チャンネル状態のチェック：場合によっては，システムはキャラクタを読み出す必要がなく，むしろキャラクタが有効かどうかを知る必要がある．このような情況では，システムはステータス・ポートの内容を読み出す．
5．コントロール情報のチャンネルへの送出：システムは，たとえばチャンネルがパリティ・エラーのチェックをできるように，チャンネルの状態を変える必要がある．
6．チャンネルからの一連のキャラクタの入力：ある終結条件が検出されるまで，キャラクタの入力を必要とする場合がある．たとえば，キャリッジ・リターンが終結条件を構成するか，あるいは一定数のキャラクタが入力される．たとえば，5つの数字がZIPコード* を構成する．I/Oドライバはチャンネルからデータを読み出す．これには，データが有効になるのを待ち，データ・ポートの情報を読み出してメモリの指定された場所にデータをセーブして，終結条件に達したかを決定するために調べることが含まれる．
7．チャンネルへの一連のキャラクタの出力：システムは，終結条件が検出されるまで，一連のキャラクタの出力を要求する．終結条件には，前もって決められたストリング最後のキャラクタの検出あるいは特定数のキャラクタの出力が含まれる．I/Oドライバは終結条件を調べる．終結条件が検出されなければ，I/Oドライバは，特定のメモリ位置からデータをロードして，チャンネルにデータを送出する．

3.1.3 出　力

8251Aは，チャンネルの出力特性を定義するために，コントロール情報を用いる．定義される必要のある出力仕様を以下に示す．

1．データ・パスの幅：8251Aは，5，6，7，あるいは8ビットよりなるデータ単位

---

* 日本では郵便番号（訳者注）

が可能である．このシステムでは，8データ・ビットが転送される．

2．データ転送速度：この場合，最大のデータ速度は9600ボーとなる．

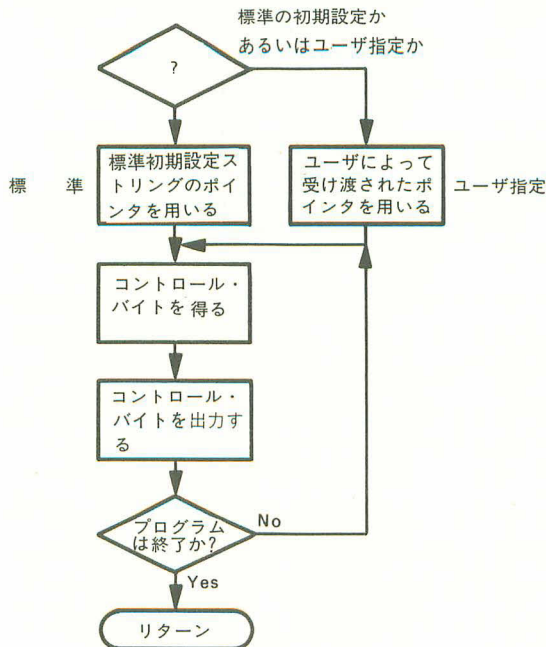
3．ハンドシェイクのプロトコル：この例では，8251Aがシステムに割り込みを用いるのではなく，システムは，8251Aが他のデータを送信できるかを判断するために，8251Aのステータス・レジスタを読み出す．

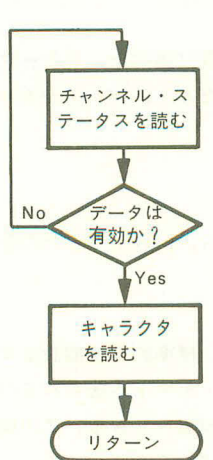
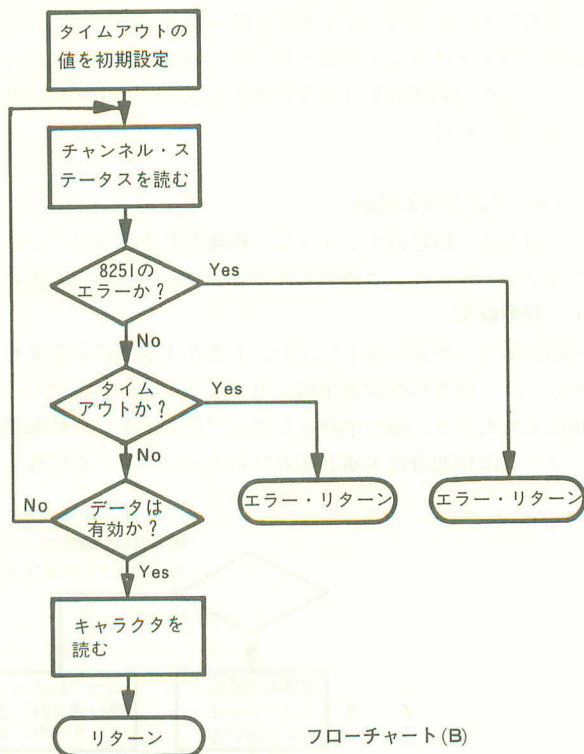
8251Aのデータとコントロール／ステータスのポートについては既に述べた．ステータス・レジスタのTxRDY（トランスミッタ・レディ）ビットが1になると，データがチャンネルに送られる．

3.1.4　プログラム設計

この項では，特定のモジュールで複雑なものはない．このことから，I/Oドライバの各々のモジュールについて述べるのに，フローチャートを用いる．

(1)　**初期設定**

初期設定ルーチンには1つだけ，主要な決定項目が含まれている．標準的な初期設定が必要ならば，標準初期設定手順に対するポインタが，コントロール・ポートへ送られるべき情報を識別する．他の手段として，“カスタム”の初期設定手順が実行できる．この場合，ユーザは初期設定手順とそれに対するポインタを用意しなければならない．

(2) 1個のキャラクタの入力

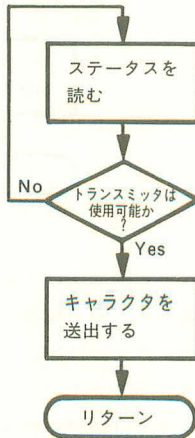
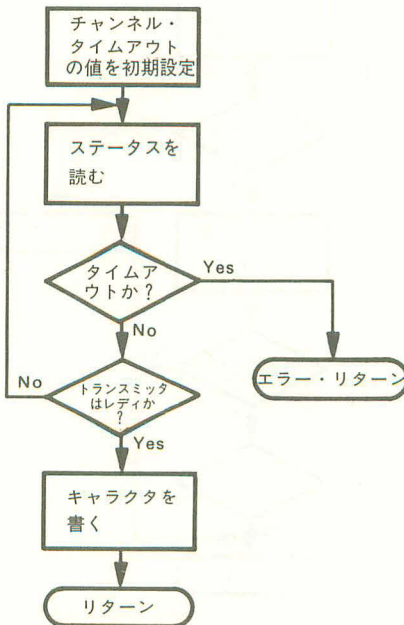
単一キャラクタ入力ルーチンのフローチャートは(A)に示されている．チャンネル・ステータスを読み出すルーチンを呼び出して，データが有効になるのを待ち，データ・ポートからデータを読み出して復帰する．

フローチャート(A)

フローチャート(B)

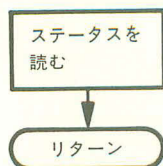
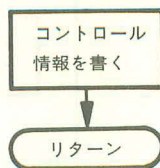
単一キャラクタ入力ルーチンの設計には，次の2つの大事な点が含まれていない．

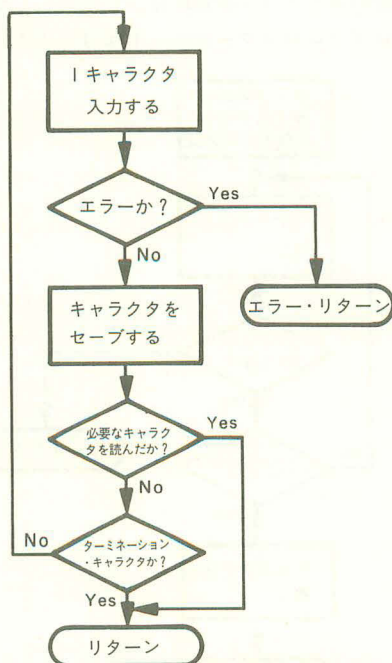
- 8251 Aのエラー処理．チャンネル・ステータスを読み出すルーチンが呼び出されると，8251 Aのステータス・レジスタのエラー・ビットが調べられる．もし8251 Aのエラーが検出されれば，単一キャラクタ入力ルーチンによってI/Oドライバに適当なエラー・コードが返される．
- タイムアウト・エラー．ドライバは，タイムアウト・クロックとして用いるために，適当なレジスタあるいはメモリ位置の初期設定を行ない，チャンネル・ステータスの読み出しを行なうルーチンを呼び出すたびに，その内容を減じる．タイムアウトのレジスタあるいはメモリ位置の内容が減少して0になれば，タイムアウトのエラー・コードを呼び出し元ルーチンへ返す．

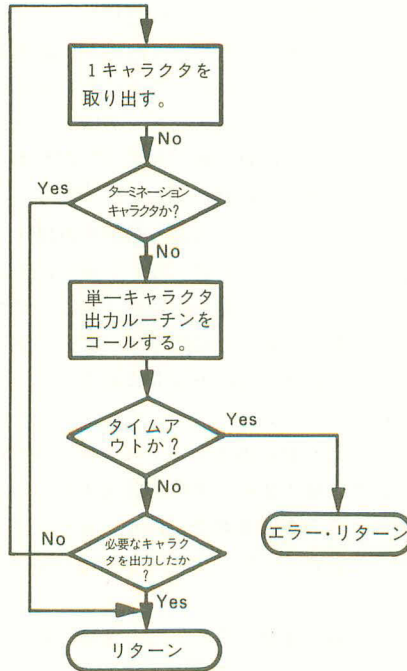
これらの考慮を単一キャラクタ入力ルーチンに加えると，フローチャートは（B）のように修正される．

(3)　1個のキャラクタの出力

単一キャラクタ出力ルーチンのフローチャートを以下に示す．チャンネル・ステータスを読み出すルーチンを呼び出して，トランスミッタが使用可能になるのを待ち，データ・ポートにキャラクタを書き込んで復帰する．

入力ルーチンと同じく，上に示したように，最初の設計にはエラーとタイムアウトについての考慮は含まれていない．8251Aの場合，ステータス・レジスタでは送信エラーが通知されないので，調べる必要のあるエラー状態は存在しない．しかし，タイムアウトのチェックは含まれるので，プログラムのフローチャートは以下のように修正される．

(4) チャンネル・ステータスのチェックとコントロール情報の送出

チャンネル・ステータスを読み出すルーチンとコントロール情報を送出するルーチンは，それぞれ簡単なフローチャートとなる．それらを以下に示す．

(5) 一連のキャラクタの入力

複数キャラクタの入力ルーチンは，データを読み込むために，単一キャラクタ入力ルーチンを用いている．単一キャラクタ入力ルーチンから復帰すると，エラーのチェックが行なわれる．単一キャラクタ入力ルーチンでエラーが検出されると，このエラーは呼び出し元ルーチンに受け渡される．呼び出し元プログラムで指定された位置にキャラクタのセーブを行なった後，複数キャラクタ入力ルーチンは終結条件を調べる．ルーチンで，指定された数のキャラクタが読み込まれたか，ターミネーション・キャラクタが読み込まれていれば，複数キャラクタ入力ルーチンは呼び出し元プログラムに復帰する．

(6) 一連のキャラクタの出力

複数キャラクタ出力ルーチンは，ユーザ指定の位置からキャラクタをロードする．キャラクタがターミネーション・キャラクタであれば，複数キャラクタ出力ルーチンは呼び出し元プログラムに復帰する．そうでなければ，キャラクタは単一キャラクタ出力ルーチンに送られる．単一キャラクタ出力ルーチンから復帰すると，タイムアウトのチェックが行なわれる．タイムアウトが検出されれば，それは呼び出し元プログラムに受け渡される．複数キャラクタ出力ルーチンは，次に指定された数のキャラクタを出力したかチェックする．もししていれば，呼び出し元プログラムに復帰する．

3.2　8086の命令セット

8086の命令セットは，16ビット・マイクロプロセッサの新しい世代を代表する複雑なものとなっている．8086の命令セットは約70の基本命令より成り，メモリ参照命令では30にも及ぶアドレッシング・モードが利用できる．

ＣＰＵの命令セットの記述には，以下の基本的な情報が含まれる必要がある。

1．ＣＰＵの構成はどのようになっているのか．すなわち，どのようなレジスタとステータスが利用できるのか．各レジスタの主要な用途は何か．

2．どのような命令が利用できるのか．明らかに，各命令の機能に関連した解説と共に，命令セットの内容豊富な一覧表が必要である．この一覧表は，いくつかある方法の1

つで構成される．この章では，個々の命令を見つけやすいように，命令をアルファベット順にあげてある．次章では，命令をバイトあるいはグループで調べられるような機能（たとえば，算術演算操作は１つの節で論じている）に従って命令をあげてある．

3．ＣＰＵはどのような型のデータを処理するのか．簡単なＣＰＵでは，すべてのデータは１つの形式で，多分バイトとして，処理されることを必要としている．より融通性のあるＣＰＵでは，個々のビット，バイト，16ビットのワード，32ビット長のワードとして，データのアドレス指定を選択できる．

4．オペランドのソースとディスティネーションにどのようなアドレス指定が選べるのか．簡単なマイクロプロセッサでは，メモリとＣＰＵのレジスタとの間でデータの移動を行なう命令によってのみ，メモリのアドレス指定が可能であり，データについてのすべての操作は，オペランドがＣＰＵのレジスタに存在することを必要とする．より複雑なマイクロプロセッサでは，１つのオペランドはメモリからフェッチすることが可能で，もう一方のオペランドはＣＰＵのレジスタに存在する．ある場合には，両方のオペランドがメモリに存在できるＣＰＵもある．メモリ・オペランドの重要性を判断するときは，可能なメモリ・アドレス指定の選択を評価する必要がある．

5．どの命令にどのようなアドレッシング・モードが可能か．特定の命令に対してどのようなアドレッシング・モードが可能かを知ることは，命令セットの有効な利用へのかぎとなる．しかし，各命令に対して可能なすべてのアドレッシング・モードを記述しようとしたならば，この本は15巻のセットにもなってしまうだろう．したがって，命令セットの一覧の前に，アドレッシング・モードの節を設けている．

6．命令の多数のグループは，ＣＰＵのステータス・レジスタにどのように影響するのか．アセンブリ言語において各種の条件の表現を評価するためには，条件がアセンブリ言語にどのように変換されるかを知る必要がある．命令がステータス・フラグにどのように影響するかを知ることによって，プログラマはアセンブリ言語で条件付き表現を書くことが可能となる．

7．ある場合には，他の情報が重要となる．たとえば，特定の命令が実行に要するサイクル数，あるいは命令が占めるプログラム・メモリのバイト数である．この場合，各命令の記述には実行に要するサイクル数が示されている．しかし，各命令が必要とするバイト数は，場合によって指定されるアドレッシング・モードに大きく依存している．

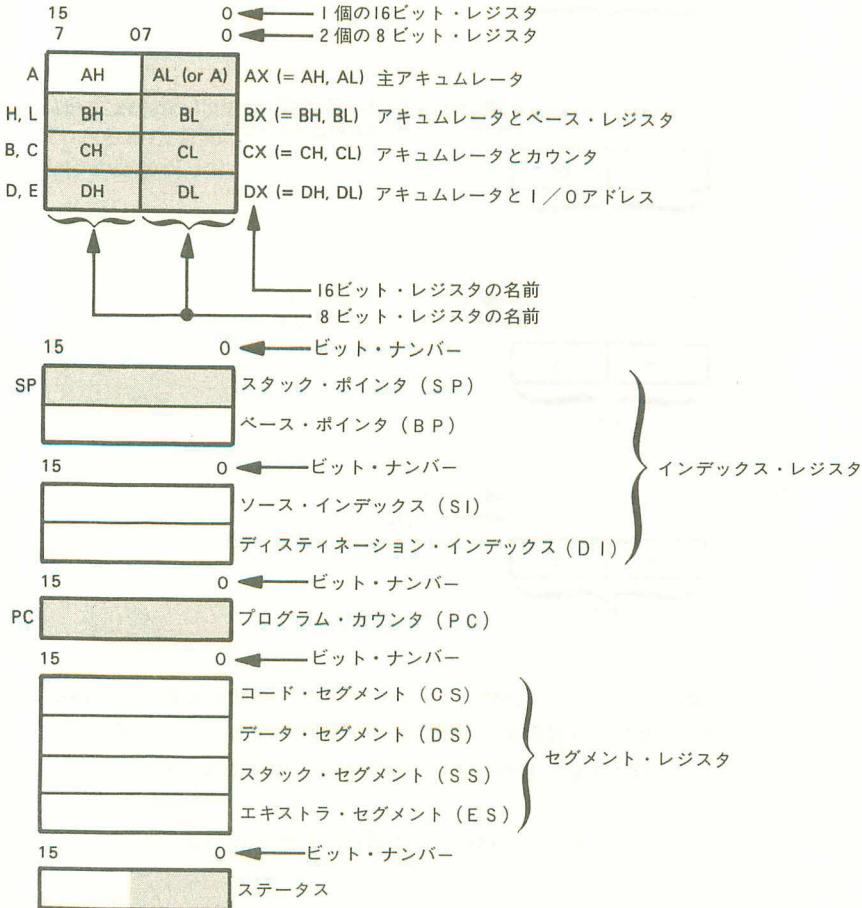
8086の命令セットのここでの議論は，以下の順序で進められている．

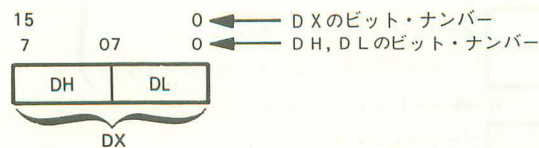
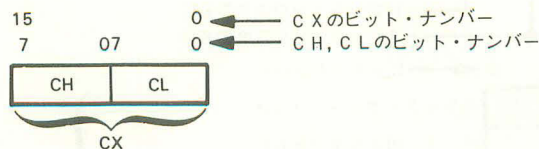
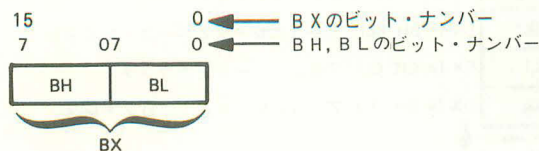
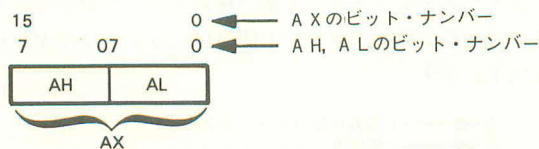
1．8086のレジスタとステータス・レジスタについてと，命令の各グループによってステータスがどのように影響を受けるかについての解説．ステータス・レジスタはまた，フラグ・レジスタあるいはプログラム・ステータス・ワードと呼ばれる．

2．8086のアドレッシング・モードについての解説．

3．8086の各命令についての解説．このセクションの前には，命令記述に用いられるシンボルと用語の要約が示されている．

3.3　8086のレジスタとフラグ

8086には，4個の16ビット汎用レジスタ，2個の16ビット・ポインタ・レジスタ，2個の16ビット・インデックス・レジスタ，1個の16ビット・プログラム・カウンタ，4個の16ビット・セグメント・レジスタ，さらに1個の16ビット・フラグ・レジスタがある．これらのレジスタを以下に示す．

3.3.1 汎用レジスタ

汎用レジスタは，2つの別々の8ビット・レジスタとして参照される．それを以下に示す．

これは，8ビット数値に，8ビット操作よりも時間とメモリ領域を必要とする16ビット操作を行なう代わりに，8ビット操作を行なえることで有利となる．たとえば，レジスタに200_{10}を初期設定して，続く事象に基づいて0に減少させる場合には，8ビットのレジスタで明らかに十分である．

汎用レジスタは，すべての8あるいは16ビットの算術演算や論理演算のオペランドとして用いることができる．

AXレジスタは主要なアキュムレータとして用いられる．このレジスタは他にはない2つの特徴を持つ．すべてのI/O操作はこのレジスタを通して行なわれ，イミディエイト・データを用いる操作が，このレジスタに対して行なわれたときは，普通必要なメモリ領域は少なくなる．さらに，ストリング操作と算術演算の命令のあるものはこのレジスタを用いる．

ALレジスタは一般に，8080AのAレジスタに対応している．

ＢＸレジスタはベース・レジスタと呼ばれる．これは，8086のメモリ・アドレスの計算に用いられる唯一の汎用レジスタである．メモリ・アドレスの計算にこのレジスタを用いるメモリ参照はすべて，デフォルト・セグメント・レジスタとしてＤＳレジスタを用いる．ＢＸレジスタは一般に，8080ＡのＨＬレジスタに対応している．すなわち，ＢＨレジスタは8080ＡのＨレジスタに，ＢＬレジスタは8080ＡのＬレジスタに対応する．

ＣＸレジスタはカウント・レジスタと呼ばれる．このレジスタは，ストリングとループの操作で減少する．ＣＸは普通ループの繰返し回数を制御するために用いられる．また，複数ビットのシフトとローテートにも用いられる．このレジスタは一般に，8080ＡのＢＣレジスタに対応している．

ＣＨレジスタは8080ＡのＢレジスタに対応している．ＣＬレジスタは8080ＡのＣレジスタに対応している．

ＤＸレジスタは，ほとんどニーモニックの理由から，データ・レジスタと呼ばれる．このレジスタはI/O命令に対してI/Oのアドレスを与える．この機能は8086の他のレジスタでは行なえない．このレジスタは一般に，8080ＡのＤＥレジスタに対応している．

ＤＨレジスタは8080ＡのＤレジスタに対応し，ＤＬレジスタは8080ＡのＥレジスタに対応する．ＤＸレジスタはまた，乗算と除算を含む算術演算操作にも用いられる．

3.3.2　ポインタ・レジスタ

ポインタ・レジスタは，スタック・セグメント内のデータにアクセスするために用いられる．それらはすべて16ビットの算術演算あるいは論理演算の操作で，オペランドとして用いられる．

ＳＰレジスタは，スタック・ポインタと呼ばれ，メモリのスタックを可能にする．メモリ・アドレス指定にＳＰを参照するものはすべて，セグメント・レジスタとしてＳＳレジスタを用いる．このレジスタは一般に，8080ＡのＳＰレジスタに対応している．

ＢＰレジスタは，ベース・ポインタと呼ばれ，スタック・セグメント中のデータをアクセス可能にする．通常は，スタックを通して受け渡されるパラメータを参照するためにこのレジスタが用いられる．

3.3.3　インデックス・レジスタ

インデックス・レジスタは，データ・メモリ内のデータにアクセスするために用いられる．インデックス・レジスタは，ストリング操作で広く用いられる．すべての16ビットの算術演算や論理演算の操作では，オペランドとして用いられている．

3.3.4　セグメント・レジスタ

セグメント・レジスタは，すべてのメモリ・アドレス指定の計算に含まれている．各セグメント・レジスタは，8086のメモリ・アドレス領域で，そのセグメント・レジスタのカレント・セグメントと呼ばれる，64キロのメモリのブロックを定義する．たとえば，ＤＳレジスタは，カレント・データ・セグメントと呼ばれる64キロのセグメントを定義する．

ＣＳレジスタは，コード・セグメント・レジスタとしても知られている．各命令のフェッチの間に，フェッチされるべき命令のメモリ・アドレスの計算のために，プログラム・カウンタの内容はＣＳレジスタの内容に加算される．

ＤＳレジスタは，データ・セグメント・レジスタとしても知られている．すべてのデータ・メモリの参照は，次の３つの例外を除いて，データ・セグメント・レジスタ相対となる．

1．スタック・アドレスは，スタック・ポインタを用いて計算される．

2．ＢＰレジスタを用いて計算されるデータ・メモリ・アドレスは，スタック・セグメント相対となる．

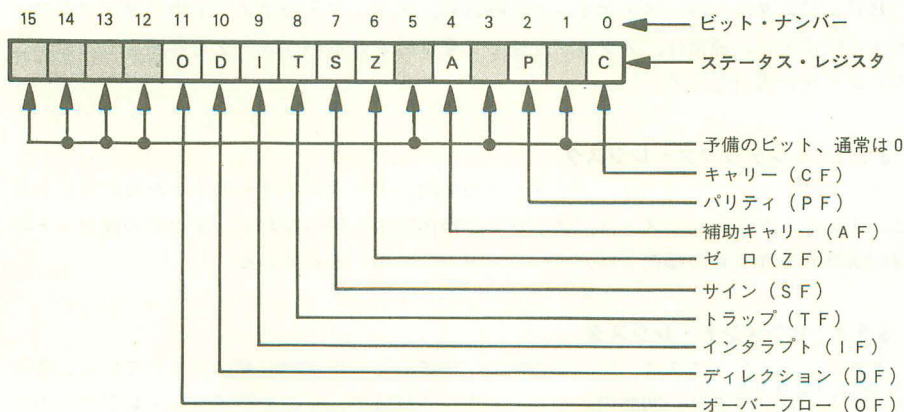
3．ストリング操作（アドレス計算にＤＩレジスタを用いるもの）は，エキストラ・セグメント相対となる．

ＳＳレジスタは，スタック・セグメント・レジスタとも呼ばれる．アドレス計算にＳＰあるいはＢＰのレジスタを用いるデータ・メモリの参照はすべて，ＳＳレジスタ相対となる．したがって，スタックに対する命令（たとえば，ＰＵＳＨ，ＰＯＰ，ＣＡＬＬ，ＲＥＴやＩＮＴなど）は，セグメント・レジスタとしてＳＳレジスタを用いる．

ＥＳレジスタは，エキストラ・セグメント・レジスタとも呼ばれる．ストリング操作は，ＥＳレジスタ相対となり，ＤＩレジスタを用いてメモリ・アドレスを計算する．

セグメント・レジスタの使用は，一般に命令によって暗黙に指定されているが，ほとんどの場合にこのセグメント・レジスタの変更を可能とする機構については後で論じる．

3.3.5 フラグ・レジスタ

8086は，ステータス・レジスタあるいはプログラム・ステータス・ワードとも呼ばれる16ビットのフラグ・レジスタを持つ．このレジスタを以下に示す．

キャリー，補助キャリー，オーバーフロー，サインのフラグは全く標準的なものである．

キャリー・フラグは，算術演算による上位ビットからのキャリーを表わす．キャリーはまたシフトやローテートの命令によっても変化する．

オーバーフロー・フラグは，算術演算による最上位ビットへのキャリーと最上位ビットからのキャリーのエクスクルーシブORをとったものである*．これは，符号付き2進演算のオーバーフローを意味する．

サイン・フラグは，算術演算操作による最上位ビットに等しい．符号付き2進演算が行なわれていると仮定すれば，サイン・フラグの0は正の結果を表わし，サイン・フラグの1は負の結果を表わす．

補助キャリーは，8080Aの同じ名前のフラグと同一である．これは，8ビットのデータ単位で，ビット3からのキャリーを表わす．

減算命令では，減数から被減数を減算するために2の補数演算を用いる．ただし，キャリー・フラグは反転している．すなわち，減算操作によって，最上位ビットからのキャリーがなければキャリー・フラグは1にセットされ，最上位ビットからのキャリーがあれば0にリセットされる．したがって，キャリー・フラグはボローを表わす．

パリティ・フラグは，データ操作の結果の下位8ビットに偶数個の1のビットがあれば1にセットされる．1のビットが奇数個あれば，パリティ・フラグは0にリセットされる．

ゼロ・フラグは，データ操作の結果が0ならば1にセットされ，データ操作の結果が0でなければ0にリセットされる．

ディレクション・フラグは，ストリング操作でインデックス・レジスタの内容を自動増加させるかあるいは自動減少させるかを決める．ディレクション・フラグが1ならば，SIとDIのインデックス・レジスタの内容は減少する．すなわち，ストリングは最上位のメモリ・アドレスから最下位のメモリ・アドレスへとアクセスが行なわれる．ディレクション・フラグが0ならば，SIとDIのインデックス・レジスタの内容は増加する．すなわち，ストリングは最下位のメモリ・アドレスから始まってアクセスされる．

インタラプト・フラグは，割り込みの有効・無効を支配する．このフラグは，8086の割り込みを有効とするためには1でなければならない．このフラグが0ならば，すべての割り込みは無効となる．

トラップ・フラグは，8086を“シングル・ステップ”のモードにする特殊なデバッグ用の助けとなる．シングル・ステップのモードは，その存在が割り込みのロジックに依存するので，8086の割り込みのロジックと共に詳細を述べる．

キャリー，補助キャリー，パリティ，サイン，さらにゼロのフラグは，8080Aにも見られる．オーバーフロー，ディレクション，インタラプト，トラップのフラグは8086に新しいものである．

---

* この結果が1のときだけ，符号付き2進演算の加減算でオーバーフローが生じている（訳者注）．

3.3.6 命令がどのようにフラグ・レジスタに作用するか

以下の一覧は，個々の命令とそれがフラグ・レジスタに与える影響について示した表を識別する．たとえば，ＡＤＤ命令のフラグへの影響については，**表3-2**を参照．

命令のニーモニック	表	命令のニーモニック	表
AAA	3-4	LODS	3-1
AAD	3-10	LOOP 命令	3-1
AAM	3-10	MOV	3-1
AAS	3-4	MOVS	3-1
ADC	3-2	MUL	3-6
ADD	3-2	NEG	3-2
AND	3-7	NOP	3-1
CALL	3-1	NOT	3-1
CBW	3-1	OR	3-7
CLC	3-9	OUT	3-1
CLD	3-9	POP	3-1
CLI	3-9	POPF	3-12
CMC	3-9	PUSH	3-1
CMP	3-2	PUSHF	3-1
CMPS	3-2	RCL	3-8
CWD	3-1	RCR	3-8
DAA	3-5	REP	3-1
DAS	3-5	RET	3-1
DEC	3-3	ROL	3-8
DIV	3-11	ROR	3-8
ESC	3-1	SAHF	3-9
HLT	3-1	SAR	3-7
IDIV	3-11	SBB	3-2
IMUL	3-6	SCAS	3-2
IN	3-1	SEG	3-1
INC	3-3	SHL	3-7
INT	3-13	SHR	3-7
INTO	3-13	STC	3-9
IRET	3-12	STD	3-9
Jump-on-Conditions (条件付分岐)	3-1	STI	3-9
JCXZ	3-1	STOS	3-1
JMP	3-1	SUB	3-2
LAHF	3-1	TEST	3-7
LDS	3-1	WAIT	3-1
LEA	3-1	XCHG	3-1
LES	3-1	XLAT	3-1
LOCK	3-1	XOR	3-7

(1) 何の影響も与えない

表3-1の命令は，8086のどのフラグにも何の影響も与えない．

(2) すべての算術演算のフラグに影響を与える

表3-2の命令は，8086の６つの算術演算のフラグ，オーバーフロー，キャリー，補助キャリー，ゼロ，サイン，パリティのすべてに影響を与える．

(3) キャリーを除くすべての算術演算のフラグに影響を与える

表3-3の命令は，8086のキャリーを除くすべての算術演算のフラグに影響を与える．オーバーフロー，補助キャリー，ゼロ，サイン，パリティはすべて影響を受ける．

(4) すべての算術演算のフラグに影響を与える（ＡＦとＣＦが意味を持つ）

表3-4の命令は，8086のすべての算術演算のフラグに影響を与える．ただし，ＡＦとＣＦの値だけが意味を持つ．オーバーフロー，ゼロ，パリティ，サインの値は不明である．

表3-1　フラグ・レジスタに何の影響も与えない命令

CALL	LOOP 命令
CBW	MOV
CWD	MOVS
ESC	NOP
HLT	NOT
IN	OUT
Jump-on-Conditions（条件付分岐）	POP
JCXZ	PUSH
JMP	PUSHF
LAHF	REP
LDS	RET
LEA	SEG
LES	STOS
LOCK	WAIT
LODS	XCHG
	XLAT

表3-2　すべての算術演算フラグに影響する命令

ADC	NEG
ADD	SBB
CMP	SCAS
CMPS	SUB

表3-3　CF を除くすべての算術演算フラグに影響する命令

DEC	INC

表3-4　AF と CF に影響する命令

AAA	AAS

(5)　すべての算術演算のフラグに影響を与える（オーバーフローは定義されない）

表3-5の命令は，8086のすべての算術演算のフラグに影響を与える．ただし，オーバーフロー・フラグは意味を持たない．キャリー，補助キャリー，ゼロ，サイン，パリティはすべて意味を持つ．

表3-5　OF を未定義とする命令

DAA	DAS

(6)　すべての算術演算のフラグに影響を与える（ＣＦとＯＦが意味を持つ）

表3-6の命令は，8086のすべての算術演算のフラグに影響を与える．キャリーとオーバーフローのフラグは，通常のようには影響を受けない．このフラグの設定についての記述は，その命令を参照のこと．他の算術演算のフラグはすべて不定となる．

表3-6　CF と OF が意味を持ち，すべての算術演算フラグに影響する命令

IMUL	MUL

(7)　すべての算術演算のフラグに影響を与える（ＡＦは定義されない）

表3-7の命令は，8086のすべての算術演算のフラグに影響を与える．キャリーとオーバーフローは0にクリヤされ，ＡＦは不定となる．ゼロ，パリティ，サインは通常どおり設定される．

表3-7　AF を未定義として，すべての算術演算フラグに影響する命令

AND	SHR
OR	TEST
SAR	XOR
SHL	

(8)ＣＦとＯＦにだけ影響を与える

表3-8の命令は，キャリーとオーバーフローのフラグにだけ影響を与える．補助キャリー，ゼロ，サイン，パリティのフラグは変化しない．

表3-8　CF と OF のみに影響する命令

RCL	ROL
RCR	ROR

(9)　特定のフラグにだけ影響を与える

表3-9の命令は，特定のフラグに作用するために用いられる．たとえば，ＳＴＩはインタラプト・フラグを１に設定するために用いられる．

表3-9　特定のフラグに影響する命令

CLC（クリア・キャリー）	SAHF（AHを8080フラグへ移動）
CLD（クリア・ディレクション）	STC（セット・キャリー）
CLI（クリア・インタラプト）	STD（セット・ディレクション）
CMC（コンプリメント・キャリー）	STI（セット・インタラプト）

⑽　パリティ，サイン，ゼロのフラグに影響を与える

表3-10の命令は，パリティ，サイン，ゼロのフラグに影響を与える．キャリー，オーバーフロー，補助キャリーのフラグは，この命令の実行によって不定となる．

表3-10　PF, SF, ZFに影響する命令

AAD	AAM

⑾　すべての算術演算のフラグを不定にする

表3-11の命令は，すべての算術演算のフラグを不定にする．

表3-11　算術演算フラグをすべて未定義にする命令

DIV	IDIV

⑿　すべてのフラグをスタックから再現する

表3-12の命令は，スタックからのデータを8086のすべてのフラグにポップする．

表3-12　スタックからすべてのフラグを復元する命令

IRET	POPF

⒀　ＩＦとＴＦにだけ影響を与える

表3-13の命令は，インタラプトとトラップのフラグをクリヤする．ＩＮＴＯ命令は，オーバーフロー・フラグが1の場合だけ，上記フラグに影響を与える．

表3-13　IFとTFをクリヤする命令

INT	INTO

ＤＩＶとＩＤＩＶの命令は，除算エラーの場合だけＩＦとＴＦに影響を与える．

3.4 8086のアドレッシング・モード

8086のアドレッシング・モードに関して，興味深い2つの主要な話題がある．

1．メモリ・アドレスがどのように形成されるか．

2．どのようなアドレッシング・モードが利用可能か．

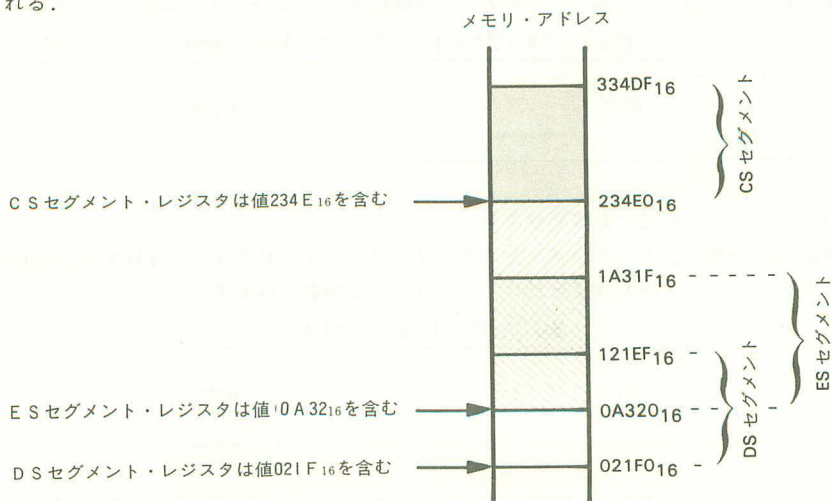
8086のメモリ・アドレスは，セグメント・レジスタの内容と有効メモリ・アドレスを加えることによって計算される．有効メモリ・アドレスは，他のマイクロプロセッサの場合と同じように，種々のアドレッシング・モードで計算される．選択されたセグメント・レジスタの内容は4ビット左へシフトされ，有効メモリ・アドレスに加えられて，次のように実アドレスの出力を生成する．

セグメント・レジスタの内容：	XXXXXXXXXXXXXXXX0000
有効メモリ・アドレス：	＋0000YYYYYYYYYYYYYYYY
実アドレスの出力：	ZZZZZZZZZZZZZZZZYYYY

X，Y，Zは2進数の1桁を表わす．

したがって20ビットのメモリ・アドレスが計算される．これは，1,048,576バイトの外部メモリを直接にアドレス指定することを可能にする．

8086のアドレスは，したがって，セグメント・アドレスと呼ばれるセグメント・レジスタの内容と，オフセット・アドレスと呼ばれる有効メモリ・アドレスの，2つの異なるアドレスで構成されている．

8086のセグメント・レジスタは，他のマイクロプロセッサのレジスタとは異なっている．これは，丁度16バイトの倍数となるアドレス境界に位置する任意のメモリ位置を示すベース・レジスタとして作用する．適当なメモリ・アドレスを用いると，これは以下のように示される．

上に示したように，各セグメント・レジスタは，65,536バイトのメモリ・セグメントの始まりを識別する．8086は4つのセグメント・レジスタを有するので，常に4つの選ばれた65,536バイトのメモリ・セグメントが存在する．実アドレスの出力は，常にこの4つのうちの1つのセグメント内のメモリ位置を選択する．たとえば，実アドレスの出力がＤＳセグメント・レジスタと有効メモリ・アドレスの和ならば，実アドレスの出力はＤＳセグメント内のメモリ位置を選択しなければならない．すなわち，前の例では $021F0_{16}$ から $121EF_{16}$の範囲のアドレスとなる．同様に，ＣＳセグメント・レジスタと有効メモリ・アドレスの和である実アドレスの出力は，前の例では $234E0_{16}$ から $334DF_{16}$の範囲のアドレスに位置するＣＳセグメント内のメモリ位置を選択しなければならない．

セグメント・レジスタの内容についての制限はない．したがって，8086のメモリは，65,536バイトのページに分割されたり，4つのセグメント・レジスタが重なりのないメモリ領域を指定しなければならないことはない．各セグメント・レジスタは，アドレス指定の可能なメモリ内の任意の場所に65,536バイトのメモリ・セグメントの原点を指定し，他のセグメントとの重なりがあってもなくてもよい．

8086のアドレッシング・モードは次の2つの異なるタイプに分けられる．

1．プログラム・メモリ・アドレッシング・モード

2．データ・メモリ・アドレッシング・モード

この2点について論じ，この節の終わりにこれが8086でどのように実行されているかを示す．

3.4.1　プログラム・メモリ・アドレッシング・モード

命令のフェッチが行なわれるときは常に，命令がフェッチされるメモリ位置のアドレスは，プログラム・カウンタ（ＰＣレジスタとも呼ばれる）から得られるオフセットとＣＳレジスタから得られるセグメントの和として計算される．通常は，命令が実行されるに従って，ＰＣレジスタの内容は増加する．ただし，ジャンプとコールの命令は，以下の3つのうちのいずれかの方法でＰＣレジスタの内容を変更する．

1．プログラム相対アドレッシング：イミディエイト・データの形で命令によって与えられる8ビットあるいは16ビットのディスプレイスメントを，符号付き2進数としてＰＣレジスタに加算する．これはＣＳレジスタの内容を変えない．したがって，これをセグメント内操作と呼ぶ．

2．ダイレクト・アドレッシング：イミディエイト・データの形で命令中に存在する2つの新しい16ビットのアドレスを，プログラム・カウンタとＣＳレジスタにロードする．これはセグメント間操作と呼ばれる．

3．インダイレクト・アドレッシング：データ・メモリ・アドレッシングを選択して（これは次に述べる），データ・メモリからデータを読み出すのに用いられる．ただし，データ入力は，ジャンプあるいはコールの命令によってメモリ・アドレスとして解釈される．2つのインダイレクト・アドレッシングの選択がある．1つの16ビット・データ・ワードが読み出される場合，プログラム・カウンタにロードされて，ジャンプ

あるいはコールは現在のCSセグメント内のメモリ位置を参照する．2つの16ビット・データ・ワードを読み出すこともできる．第1のものはプログラム・カウンタにロードされて，第2のものはCSセグメント・レジスタにロードされる．したがって，インダイレクト・アドレッシングを用いて，アドレス指定可能なメモリ位置にジャンプあるいはコールすることができる．

3.4.2 データ・メモリ・アドレッシング・モード

8086は広範な種類のアドレッシングを持つ．これを次の6つの基本的分類に要約する．

1．イミディエイト

2．ダイレクト

3．ダイレクト・インデックス修飾

4．暗黙指定

5．ベース相対

6．スタック

(1) イミディエイト・メモリ・アドレッシング

このアドレッシングの形式では，オペランドの1つは命令のオブジェクト・コード（オペコード）直後のバイトに存在する．オペコードの次にアドレッシング・バイトがあれば，イミディエイト・データはそれに続いている．たとえば，

ADD　　AX，3064H

はアセンブラに，AXレジスタに3064_{16}を加えるADD命令を生成することを求める．これは次のように示される．

x，y，m，p，nはすべて16進数を表わす

16ビットのイミディエイト・データは，プログラム・メモリにストアされているとき，下位のバイトが上位のバイトに先行していることに注意．これは，8080Ａがイミディエイト・オペランドをプログラム・メモリにストアする方法と一致している．さらに，8086が16ビットのオペランドをデータ・メモリにストアする方法と一致している．16ビットのストアが行なわれるときは，データの下位8ビットが下位のメモリ・バイトにストアされ，データの上位8ビットが続くメモリ・バイトにストアされる．

この例では，ＡＸへのＡＤＤの命令のオペコードにすぐ続く2バイトが，ＡＸレジスタに加算される．

(2)　ダイレクト・メモリ・アドレッシング

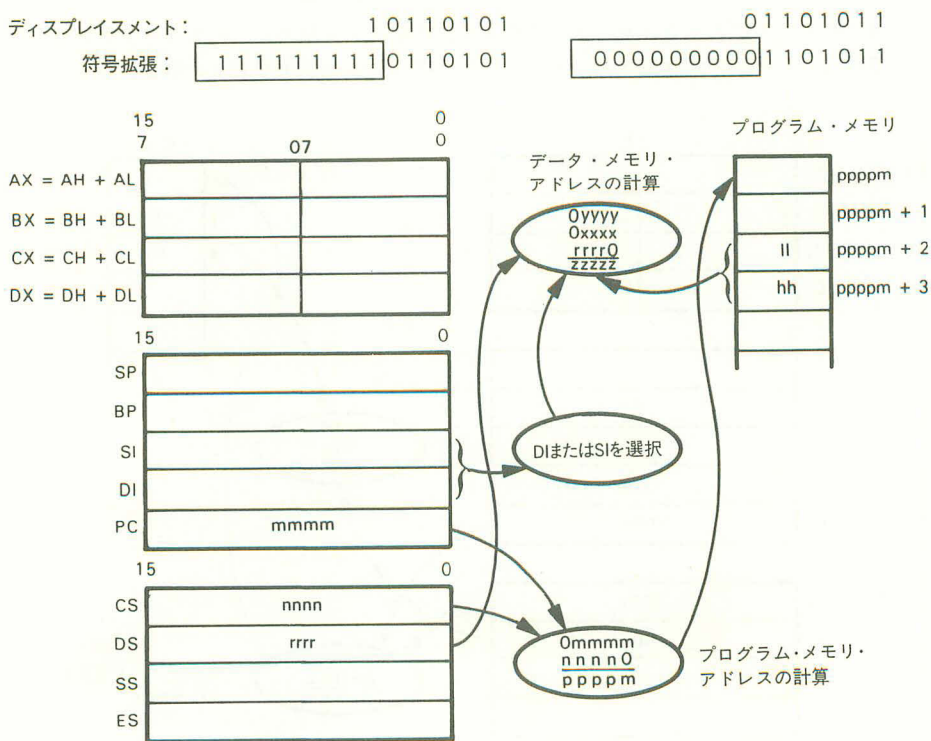
8086は，オブジェクト・コードの2バイトで与えられる16ビットのディスプレイスメントをデータ・セグメント・レジスタに加えて，容易にダイレクト・メモリ・アドレッシングを行なう．上記の和が実メモリ・アドレスとなる．これは次のように示される．

＊ダイレクト・メモリ・アドレッシングに対する実際のデータ・メモリ・アドレス

16ビットのアドレス・ディスプレイスメントは，プログラム・メモリに格納されているとき，下位のバイトが上位のバイトに先行していることに注意．これは，8080Ａがアドレスをプログラム・メモリに格納する方法に一致している．

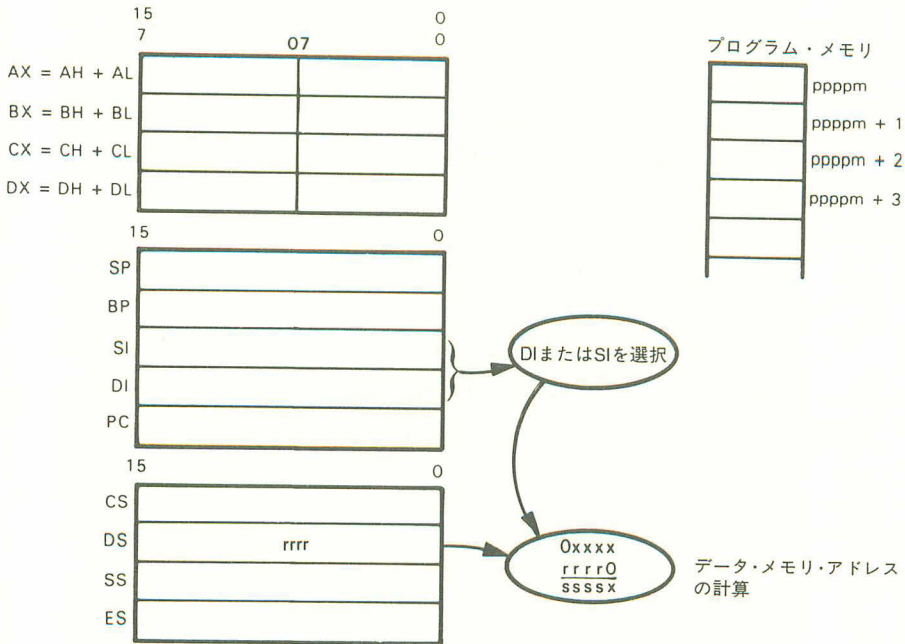
ＤＳレジスタは，上の例で示したように，データ・メモリを直接にアドレス指定するときは，セグメントのベース・アドレスでなければならない．

(3) ダイレクト・インデックス修飾メモリ・アドレッシング

ダイレクト・インデックス修飾アドレッシングは，インデックス・レジスタとしてＳＩあるいはＤＩのレジスタを指定することによって行なわれる．有効アドレスを生成するために，指定されたインデックス・レジスタの内容に，８ビットあるいは16ビットのディスプレイスメントの加算を指定できる．16ビットのディスプレイスメントはオブジェクト・コードの２バイトにストアされ，ダイレクト・メモリ・アドレッシングで示したように，ディスプレイスメントの下位バイトが上位バイトに先行している．８ビットのディスプレイスメントが指定されたならば，16ビットのディスプレイスメントを得るために，下位バイトの最上位ビットが上位バイトに拡張される．これを次に示す．

yyyyはプログラム・メモリより得られる16ビットあるいは８ビットのディスプレイスメント
xxxxはDIあるいはSIのレジスタから得られるインデックスの値

(4) 暗黙指定メモリ・アドレッシング

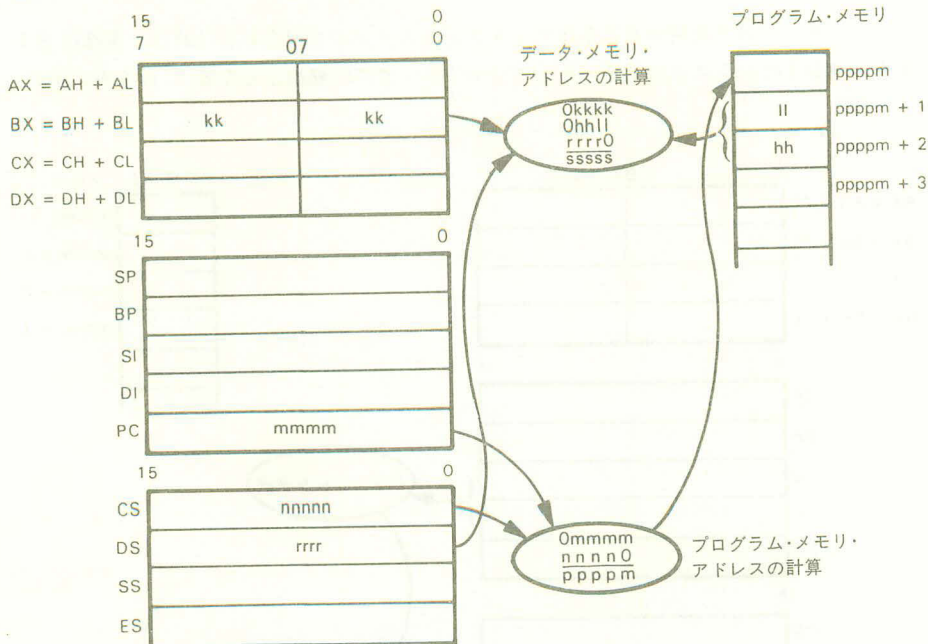
8086で暗黙指定メモリ・アドレッシングは，ダイレクト・インデックス修飾メモリ・アドレッシングの縮小されたものとして実行される．ダイレクト・インデックス修飾のアド

レッシング・モードを用いるときにディスプレイスメントを指定しなければ，実際はＳＩあるいはＤＩのレジスタによるメモリ・アドレッシングの暗黙指定となる．これを次に示す．

（別に1バイトの命令を実行することによって，DSをCS, SSあるいはESに代えられる．）

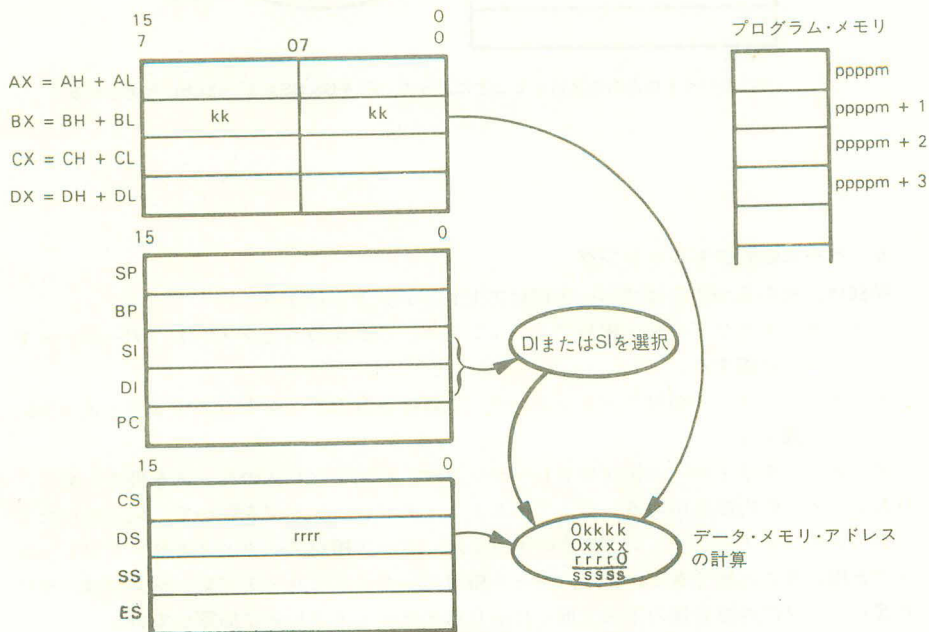
(5)　ベース相対アドレッシング

8086は，次の２つの方法でベース相対アドレッシングを行なう．

- データ・メモリ・ベース相対アドレッシング．これはＤＳセグメント（データ・メモリ）内に位置する．
- スタック・ベース相対アドレッシング．これはＳＳセグメント（スタック・メモリ）内に位置する．

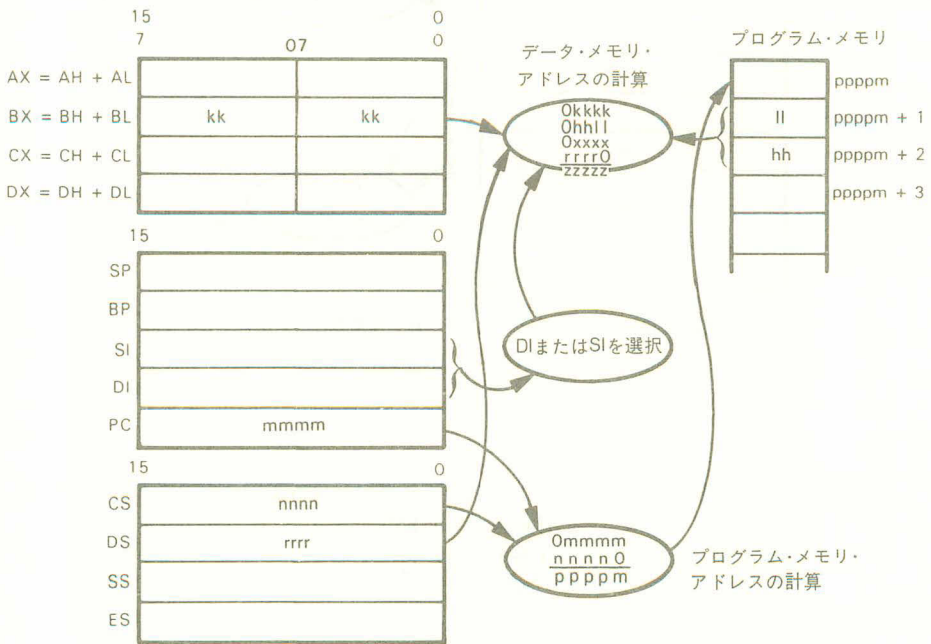
データ・メモリ・ベース相対アドレッシングは，有効アドレスのベースを得るために，ＢＸレジスタの内容を用いる．イミディエイト・アドレッシングを除いて，以上述べたデータ・メモリ・アドレッシング指定のすべては，ベース相対データ・メモリ・アドレッシングと用いることができる．実際，ベース相対データ・メモリ・アドレッシングは，単にＢＸレジスタの内容を他の方法で得られる有効メモリ・アドレスに加算している．ここで例として，ベース相対ダイレクト・アドレッシングを(A)に示す．

（別に１バイトの命令を実行することによって，DSをCS，ESあるいはSSに代えられる．）

(A)　ベース相対ダイレクト・アドレッシング

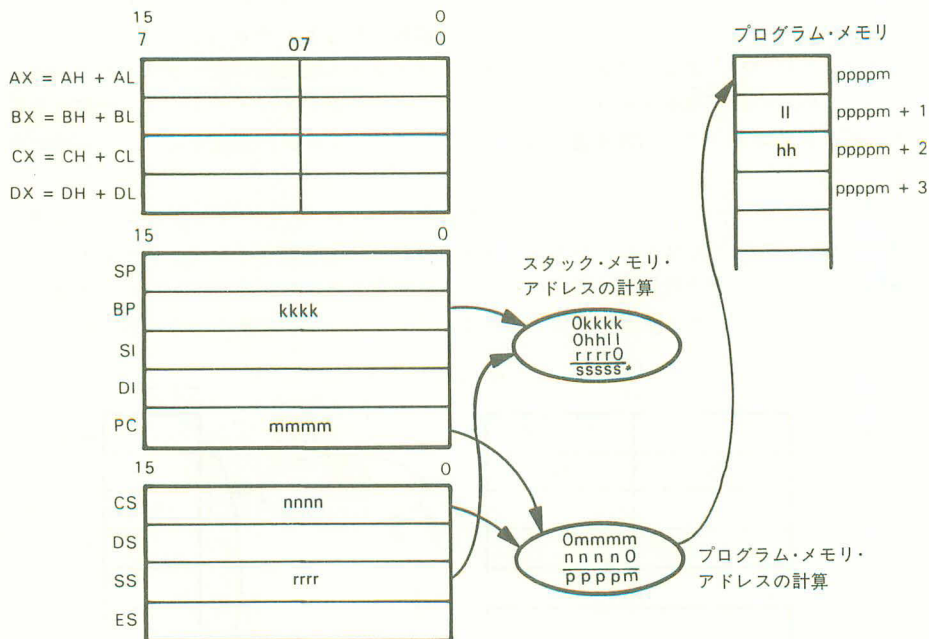
15
7
07
0
0
AX = AH + AL
BX = BH + BL
CX = CH + CL
DX = DH + DL
kk
kk
15
0
SP
BP
SI
DI
PC
15
0
CS
DS
SS
ES
rrrr
プログラム・メモリ
ppppm
ppppm + 1
ppppm + 2
ppppm + 3
DIまたはSIを選択
0kkkk
0xxxx
rrrr0
sssss
データ・メモリ・アドレス
の計算

(B)　ベース相対暗黙指定アドレッシング

前に述べた単純なダイレクト・アドレッシングは，常に16ビットのディスプレイスメントを生成する．ベース相対のダイレクト・アドレッシングでは，前の例でhhllと示されている16ビットのディスプレイスメントは，符号拡張が行なわれる8ビットのディスプレイスメントを持つか，あるいは全くディスプレイスメントを持たないことが許されている．

ベース相対の暗黙指定メモリ・アドレッシングは，有効メモリ・アドレスを計算するため，単にBXレジスタの内容を選ばれたインデックス・レジスタに加算する．これを(B)に示す．

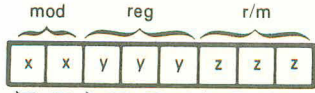
ベース相対のダイレクト・インデックス修飾データ・メモリ・アドレッシングは複雑に見えるが，実はそうではない．BXレジスタの内容を，通常のダイレクト・インデックス修飾アドレッシングで計算される有効アドレスに加算する．したがって，ベース相対のダイレクト・インデックス修飾データ・メモリ・アドレッシングは次のように示される．

上の例で，インデックスの値xxxxは選択可能である．ベース相対のダイレクト・メモリ・アドレッシングも利用可能である．この場合は，SIとDIのレジスタはどちらもアドレス計算に寄与しないので，上の例から0xxxxは取り除かれる．

(6)　スタック・メモリ・アドレッシング

8086はまた，スタック・メモリ・アドレッシングにも今述べたベース相対のデータ・メモリ・アドレッシングの選択がある．ただしこの場合，ベース・レジスタとしてBPレジスタが用いられる．ここで例として，ベース相対のダイレクト・スタック・アドレッシングを示す．

＊ベース相対直接メモリ・アドレッシングに対する実際のスタック・メモリ・アドレス出力

上の例では，16ビットのディスプレイスメントあるいは符号拡張される8ビットのディスプレイスメントとして，ディスプレイスメントhhllがある．ベース相対のスタック・メモリ・アドレッシングでは，たとえ0であっても，ディスプレイスメントを指定する必要がある．

3.4.3 アドレッシング・モード・バイト

8086では，明らかに広範なアドレッシング・モードの中から選択ができる．次に起きる疑問は，このアドレッシング・モードがオブジェクト・コードでどのように実現されるかである．8086は，ほとんどのアドレッシング・モードを，命令のオブジェクト・コードに，アドレッシング・モード・バイトとして知られる1バイトのオブジェクト・コードを用いて指定する．アドレッシング・モード・バイトは，さらに関連のある1あるいは2バイトのディスプレイスメントを持てる．最初のオブジェクト・コードの前にプレフィックス命令が含まれていなければ，アドレッシング・モード・バイトは常に命令オブジェクト・コードの2番目のバイトとなる．アドレッシング・モード・バイトは次のように示される．

xxはmodフィールドを構成する2ビット．modフィールドはメモリとレジスタのアドレッシングを区別するために用いられる．またメモリ・アドレッシングの場合，アドレッシング・モード・バイトに続くディスプレイスメントのバイト数を指定する．

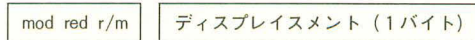
yyyはregフィールドを構成する3ビット．regフィールドは操作に用いられるレジスタを定義する．さらに，この3ビットは，後で論じられているように，命令を表わすのにも用いられる．

zzzはr／mフィールドを構成する3ビット．r／mフィールドはmodフィールドと共にアドレッシングを表わすために用いられる．

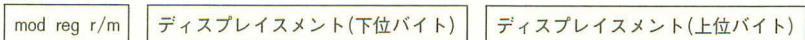
mod ＝

00　メモリ・アドレッシング・モード．r/mは厳密なアドレッシング選択を指定する．ディスプレイスメント・バイトは存在しない．

01　メモリ・アドレッシング・モード．r/mは厳密なアドレッシング選択を指定する．1バイトのディスプレイスメントが存在する．このディスプレイスメント・バイトは，＋127から－128の範囲の符号付き数値とみなされる．この数値がメモリ・アドレスの計算に用いられるときは，16ビットに符号拡張される．この場合，アドレッシング・モード・バイトは次のように表わせる．

mod red r/m	ディスプレイスメント（1バイト）

ここで，mod＝01でディスプレイスメントは8ビットの符号付き数値である．

10　メモリ・アドレッシング・モード．r/mはアドレッシング選択を指定する．2バイトのディスプレイスメントが存在する．最初のバイトはディスプレイスメントの下位8ビットで，2番目のバイトはディスプレイスメントの上位8ビットである．この数がメモリ・アドレスの計算に用いられるときは，符号なしの16ビット数値として取り扱われる．この場合，アドレッシング・モード・バイトは次のように表わせる．

mod reg r/m	ディスプレイスメント(下位バイト)	ディスプレイスメント(上位バイト)

ここで，mod＝10である．

11　レジスタ・アドレッシング・モード．r/mはレジスタを指定する．wビットと共に用いられて，8あるいは16ビットのレジスタ選択を決める．

reg　操作に使用されるレジスタの選択に，wビットと呼ばれるもう1つのビットと共に用いられる．命令のオペコードの一部であるwビットは，8あるいは16ビットのどちらが行なわれるかを選択する．

reg	w = 0	w = 1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

1個のレジスタあるいはメモリのオペランドを必要とする命令（NOT，NEGなど）や，1個の暗黙指定のオペランドを持つ命令（MOV イミディエイト，DIV，MULなど）や，目的アドレスの計算にアドレッシング・モードを用いる命令（JMP，CALLなど）は，必要な命令を指定するために，regフィールドをオペコードのバイトの拡張として用いる．例としては，ADC命令の項を参照．

r/m　　以下に示すように，modと共にアドレッシング・モードを指定する．

r/m	mod - 00	mod - 01	mod - 10	mod - 11	
				w = 0	w = 1
000	BX + SI	BX + SI + DISP	BX + SI + DISP	AL	AX
001	BX + DI	BX + DI + DISP	BX + DI + DISP	CL	CX
010	BP + SI	BP + SI + DISP	BP + SI + DISP	DL	DX
011	BP + DI	BP + DI + DISP	BP + DI + DISP	BL	BX
100	SI	SI + DISP	SI + DISP	AH	SP
101	DI	DI + DISP	DI + DISP	CH	BP
110	ダイレクト・アドレス	BP + DISP	BP + DISP	DH	SI
111	BX	BX + DISP	BX + DISP	BH	DI

この表は，ダイレクト・アドレスを除いて自明である．modが00でr/mが110のとき，オフセット・アドレスは，アドレッシング・モード・バイトに続く2バイトから直接に得られる．これは次のように示される．

mod reg r/m	オフセット・アドレス(下位バイト)	オフセット・アドレス(上位バイト)

ここで，modは00でr/mは110である．

3.4.4 セグメント変更

すべてのアドレッシング・モードは，標準的なデフォルト・セグメント・レジスタを持つ．ほとんどの場合，セグメント変更プレフィックスを用いて，他のセグメント・レジスタを選ぶことができる．プレフィックスを用いるために，デフォルト・セグメント・レジスタの指定を無効にする命令の前に，次のバイトを置く．

rrは命令で用いられるセグメント・レジスタを選択する2ビット
rr＝00：ESレジスタ
rr＝01：CSレジスタ
rr＝10：SSレジスタ
rr＝11：DSレジスタ

次の3つの場合，セグメント変更はできない．

1．オフセットの計算にスタック・ポインタ（ＳＰレジスタ）を用いるスタック参照命令（たとえば，ＰＵＳＨとＣＡＬＬ）は，常にセグメント・レジスタとしてＳＳレジスタを用いる．
2．ＤＩレジスタを用いるストリング命令は，常にセグメント・レジスタとしてＥＳレジスタを用いる．ＳＩとＤＩの両方が用いられるストリング操作(たとえば，ＭＯＶＳやＣＭＰＳ）では，セグメント変更プレフィックスが存在するならば，ＳＩのオフセットのセグメント・レジスタを無効にする．
3．セグメント変更プレフィックスは，プログラム・メモリ・アドレッシングと共には使用できない．すべての命令フェッチは，ＣＳセグメント・レジスタ相対である．

3.4.5　メモリ・アドレッシング・テーブル

メモリ・アドレッシング・モードとメモリ・アドレッシング・バイトの情報を合わせると，次のようにまとめられる．

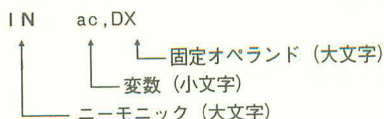
r/m =	mod = 00	mode = 01	mod = 10
000	ベース相対インデックス BX + SI	ベース相対ダイレクト・インデックス BX + SI + DISP	ベース相対ダイレクト・インデックス BX + SI + DISP
001	ベース相対インデックス BX + DI	ベース相対ダイレクト・インデックス BX + DI + DISP	ベース相対ダイレクト・インデックス BX + DI + DISP
010	ベース相対インデックス・スタック BP + SI	ベース相対ダイレクト・インデックス・スタック BP + SI + DISP	ベース相対ダイレクト・インデックス・スタック BP + SI + DISP
011	ベース相対インデックス・スタック BP + DI	ベース相対ダイレクト・インデックス・スタック BP + DI + DISP	ベース相対ダイレクト・インデックス・スタック BP + DI + DISP
100	暗黙指定 SI	ダイレクト・インデックス SI + DISP	ダイレクト・インデックス SI + DISP
101	暗黙指定 DI	ダイレクト・インデックス DI + DISP	ダイレクト・インデックス DI + DISP
110	ダイレクト・アドレス	ベース相対ダイレクト・スタック BP + DISP	ベース相対ダイレクト・スタック BP + DISP
111	ベース相対 BX	ベース相対ダイレクト BX + DISP	ベース相対ダイレクト BX + DISP

2つのオペランドを持つ命令は，1つのオペランドをメモリから，もう一方のオペランドをCPUのレジスタにしてアクセスすることが非常に多い．また，CPUのレジスタを両方のオペランドとしてアクセスすることも多い．8086は，いくつかの特殊なデータ・ストリング操作命令を除いて，両方のオペランドをメモリとすることは許されていない．次の選択が可能である．

ソース・オペランド	ディスティネーション・オペランド	結　果
CPUレジスタ	CPUレジスタ	CPUレジスタ
メモリ位置	CPUレジスタ	CPUレジスタ
CPUレジスタ	メモリ位置	メモリ位置

3.5　命令セットのニーモニック

次の節で，8086アセンブリ言語の各命令について述べる．記述の形式は，次の6つの異なる部分から成る．

1．命令のニーモニックとそれに関連した種々のオペランド．変化するオペランドは小文字で表わされている．ニーモニック自身と固定しているオペランドは大文字で表わされている．次に例を示す．

2．命令動作の記述．

3．コード化された命令の機械語．

4．命令動作の例．非常に簡単な命令では存在しない場合がある．

5．命令実行のダイアグラム．命令が8086のフラグ，レジスタ，メモリに与える影響を示す．

6．命令の使用法の簡単な例や，特定の場合にはより有効となる関連のある命令など，類別した情報を含む注釈部分．

3.5.1　略　語

ニーモニックと共に，オペランドに用いられる略語を次に示す．

ac　　8ビット操作が指定されればALレジスタを，あるいは16ビット操作が指定されればAXレジスタを表わす．これは，8086アセンブリ言語の命令では，ALあるいはAXと表わされる．

addr　　16ビットのオフセット・アドレスと16ビットのセグメント・アドレスから成る8086のアドレスである．普通これは，8086アセンブリ言語の命令で，ラベルで表わされる．

count　　1あるいはCLレジスタの内容を表わす．これは，8086アセンブリ言語の命

令では，１あるいはCLで示される．

data　８あるいは16ビットのイミディエイト・データを示す．これは，8086アセンブリ言語のステートメントで，自由な数値表現あるいは式として用いられる．

disp　ジャンプと条件付きジャンプで用いられる８ビットの符号付き２進数のディスプレイスメントを示す．常にこれは，8086アセンブリ言語の命令では，ラベルで表わされる．

disp16　コール，ジャンプ，リターンの命令で用いられる16ビット２進数のディスプレイスメントを示す．コールとジャンプの命令で用いられるとき，これはほとんど常に，ラベルで表わされる．リターン命令は普通，これを表わすために数式を用いる．リターン命令との使用は後で示す．

mem　メモリ・オペランドを示す．オペランドを選ぶためのアドレッシング・モードは，アドレッシング・モード・バイトで指定される．これは一般に，アセンブラが適当なアドレッシング・モード・バイトを選ぶラベル，あるいは特定のアドレッシング・モード・バイトの選択が可能な一連のシンボルで表わされる．

mem/reg　メモリあるいはレジスタのオペランドを示す．memとregの記述を参照．

port　I/Oポートを示す．これは，数値表現あるいは式で表わされる．ポート番号は，00_{16}からFF_{16}の間でなければならない．

reg　８ビット操作が指定されたときは，AH, AL, BH, BL, CH, CL, DH, あるいはDLのレジスタを，16ビット操作が指定されたときは，AX，BX，CX，DX，SP，BP，SI，あるいはDIのレジスタを表わす．

segreg　CS，DS，ES，あるいはSSのレジスタを表わす．

命令のコード化を示す際に用いられる略語を次に示す．

c　シフトとローテートの命令において，実行されるシフトあるいはローテートの数となる，１あるいはCLレジスタの内容を選択するために用いられる１ビットを示す．

c＝0	１ビットのシフトあるいはローテートを行なう．
c＝1	CLレジスタで指定された回数，シフトあるいはローテートを行なう．

d　操作の行なわれる方向を指定するために用いられる１ビットを示す．

disp　ジャンプと条件付きジャンプの命令で，符号付き２進数のディスプレイスメントとして用いられる８ビットを示す．

jj　イミディエイト・データあるいは16ビットのディスプレイスメントの一部を表わすために用いられる２桁の16進数．

kk　イミディエイト・データあるいは16ビットのディスプレイスメントの一部を表わすために用いられる２桁の16進数．

mod reg r/m　前にこの章で述べた8ビットのアドレッシング・モード・バイト.

rrr　3ビットで，8086の汎用レジスタの1つを選択する.

8ビット操作指定	16ビット操作指定
rrr＝000：AL	rrr＝000：AX
001：CL	001：CX
010：DL	010：DX
011：BL	011：BX
100：AH	100：SP
101：CH	101：BP
110：DH	110：SI
111：BH	111：DI

s　イミディエイト・データの，符号拡張を行なうかどうかを示す1ビット．イミディエイト・データとの16ビット操作が指定されているならば，イミディエイト・オペランドをプログラム・メモリ領域の1バイトを用いて表わすことができる．sは次のように解釈される.

s＝0　イミディエイト・データに2バイト必要．符号拡張は行なわれない.

s＝1　1バイトのイミディエイト・データが存在する．操作に必要な16ビットのイミディエイト・データを得るために，イミディエイト・データ・バイトの最上位ビットを符号拡張する.

ss　8086のセグメント・レジスタを選択する2ビット.

ss＝00：ES
01：CS
10：SS
11：DS

v　ソフトウエア・インタラプトのベクタ位置を示す1ビット．v＝0ならば，インタラプト・サービス・ルーチンはメモリ位置 $0000C_{16}$ で指定されるアドレスに位置している．それ以外では，アドレスは後続のバイトによって決定される.

w　8あるいは16ビットの操作の，どちらが行なわれるかを示す1ビット.

w＝0　8ビット操作
w＝1　16ビット操作

xxx　3つのドント・ケア* のビットを示す.

yy　命令で用いられるI/Oポート番号を示す2桁の16進数.

---

* don't care：無関係のもの（訳者注）.

以下のシンボルは，命令を用いた例で使用される．

H　　数字が16進数として処理されることを指定するために，数字の最後に用いられる．

〔 〕　　かっこの中の式で示されるメモリ位置の内容を表わすために用いられる．BXレジスタが054A $_{16}$ を含むと仮定する．

〔BX〕

の表現は，カレント・データ・セグメントで054A $_{16}$ のオフセット・アドレスを持つメモリ位置を参照している．

g, h, j, k, m, n, p, q, r, s, t, u, v, w, x, y, z

すべて16進数の1桁を表わすのに用いる．たとえば，

jjkk

は16ビットのデータ要素を表わすために用いられ，

ppppm

は20ビットのアドレスを表わすために用いられる．

EA　　有効アドレスを示す．これは，個々の命令によって必要な実行サイクル数の計算に現われる．EAは，アドレッシング・モードの実行サイクルを指定し，次のように加算される必要がある．

ダイレクト・アドレッシング	＋6サイクル
ダイレクト・インデックス修飾アドレッシング	＋9サイクル
暗黙指定アドレッシング	＋5サイクル
ベース相対アドレッシング	＋5サイクル
ベース相対ダイレクト・アドレッシング	＋9サイクル
ベース相対インデックス修飾アドレッシング	＋7あるいは＋8サイクル*
ベース相対ダイレクト・インデックス修飾アドレッシング	＋11あるいは＋12サイクル*

次の場合さらにアドレッシング・モード・サイクルの加算が必要となる．

セグメント変更プレフィックスがあるとき	＋2サイクル
16ビットのワードが指定されて，そのワードが奇数のメモリ・アドレスに位置するとき	＋4サイクル

* BP＋SIとBX＋DIのモードは，BP＋DIとBX＋SIのモードよりも1つ多くのクロックが必要．

3.6 8086アセンブリ言語の命令
(アルファベット順)

AAA (ASCII Adjust for Addition)

ASCII 加算結果のアジャスト*を行なう.

この命令は, オペランドとして2つのASCIIキャラクタの加算によって結果が得られたものとして, ALレジスタの結果を補正するために用いられる. この補正(アジャストメント)は次のようにして行なわれる.

1. ALレジスタの下位4ビットが0と9の間で, しかもAFフラグが0ならば, ステップ3へ.
2. ALレジスタの下位4ビットがAとFの間, あるいはAFフラグが1ならば, ALレジスタに6を加え, AHレジスタに1を加えて, AFフラグを1にセットする.
3. ALレジスタの上位4ビットをクリヤする.
4. CFフラグにAFフラグの値を設定する.

命令コードを次に示す.

```
AAA
 37
```

たとえば, AXレジスタの内容を 0535_{16}, BLレジスタの内容を 39_{16} とすると,

```
ADD    AL,BL
AAA
```

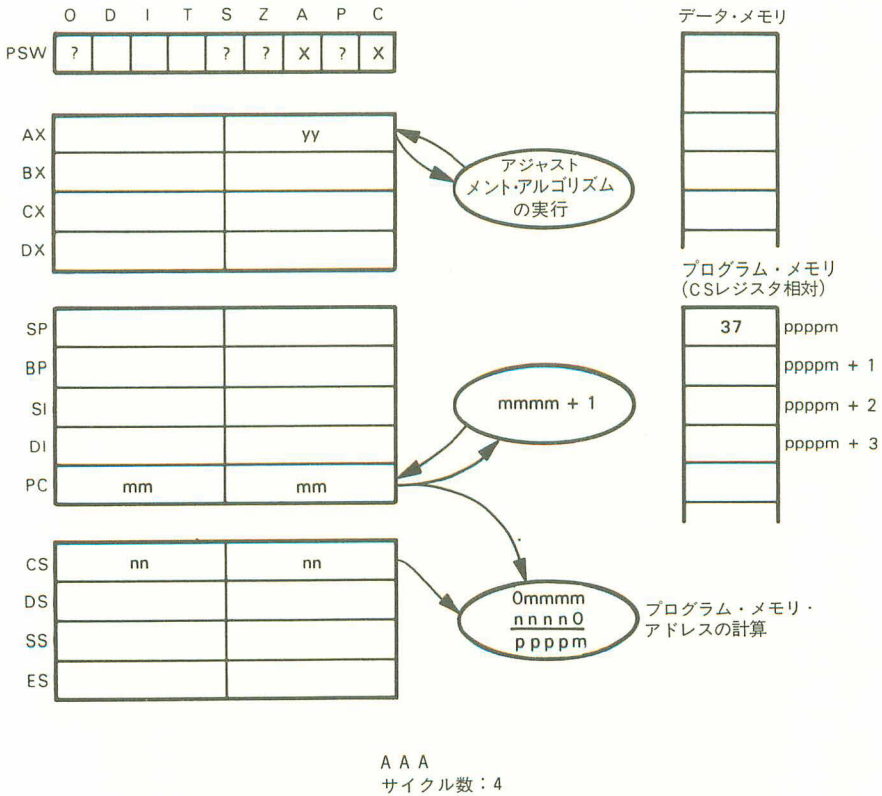
を実行することによって, AXの内容は 0604_{16} となる. ADD命令の結果,

$$
\begin{array}{rl}
35_{16}= & 0011\ \ 0101 \\
39_{16}= & 0011\ \ 1001 \\
\hline
 & 0110\ \ 1110
\end{array}
$$

となり, $6E_{16}$ がALにストアされる. AAA命令によるアジャストメント・アルゴリズムの結果, AFとCFのフラグは1にセットされ, 04_{16} がALレジスタにストアされ, そしてAHレジスタは増加して 06_{16} となる.

---

* 補正, 修正, 調整 (訳者注).

AAA
サイクル数：4

注）

1．2つのオペランドが1桁のBCDである場合にも，この命令が使えることに注意．この種の操作がなぜ必要とされるかについては読者に任せる．

2．2つのパック形式BCD加算についての補正は，DAA命令を参照．

3．この命令の結果，OF，PF，SF，ZFの各フラグは不定となる．

AAD　(ASCII Adjust for Division)

除算のためにAXレジスタのアジャストを行なう．

この命令では，AHとALのレジスタがアンパック形式BCDオペランドを含んでいることを仮定している．この命令は，前記情報をALレジスタの2進数オペランドに変換する．変換のアルゴリズムでは，10の位はAHレジスタに，1の位はALレジスタにあることを前提としている．次にAADのアルゴリズムを示す．

1．AHレジスタの内容に$0A_{16}$を乗じる．

2．ＡＨをＡＬに加算する．

3．ＡＨレジスタに00_{16}をストアする．

4．次のようにフラグを設定する．

　　ＣＦ，ＯＦ，ＡＦ：不定

　　ＰＦ，ＺＦ，ＳＦ：ＡＬレジスタに基づいて決定

命令コードを次に示す．

AAD
D50A

ＡＸレジスタが0604_{16}を含むとすると，

AAD

の実行の結果，ＡＸレジスタは0040_{16}となる．フラグは次のように設定される．

ＣＦ，ＯＦ，ＡＦ：不定

ＳＦ：ＡＬレジスタの最上位ビットが0なので，ＳＦ＝0

ＺＦ：ＡＬレジスタは0でないので，ＺＦ＝0

ＰＦ：ＡＬレジスタの1のビットは1個なので，ＰＦ＝0

AAD
サイクル数：60

注)

1．この命令は，除算のASCIIオペランドをアジャストするためにも使用可能．たとえば，AXレジスタが3537_{16}を含む場合を考えると，

```
AND     AX,0F0FH
AAD
```

の実行後，AXレジスタは0039_{16}となる．

AAM　(ASCII Adjust for Multiplication)

BCD乗算結果のアジャストを行なう．

この命令は，オペランドとして2つのアンパック形式BCDによる乗算が実行されたことを前提に，ALレジスタの結果を補正する．アジャストメントは次のように行なわれる．

1．ALレジスタを$0A_{16}$で割り，商をAHレジスタに，余りをALレジスタにストアする．
2．以下のようにフラグを設定する．
   CF，OF，AF：不定
   PF，SF，ZF：ALレジスタに基づいて決定

命令コードを次に示す．

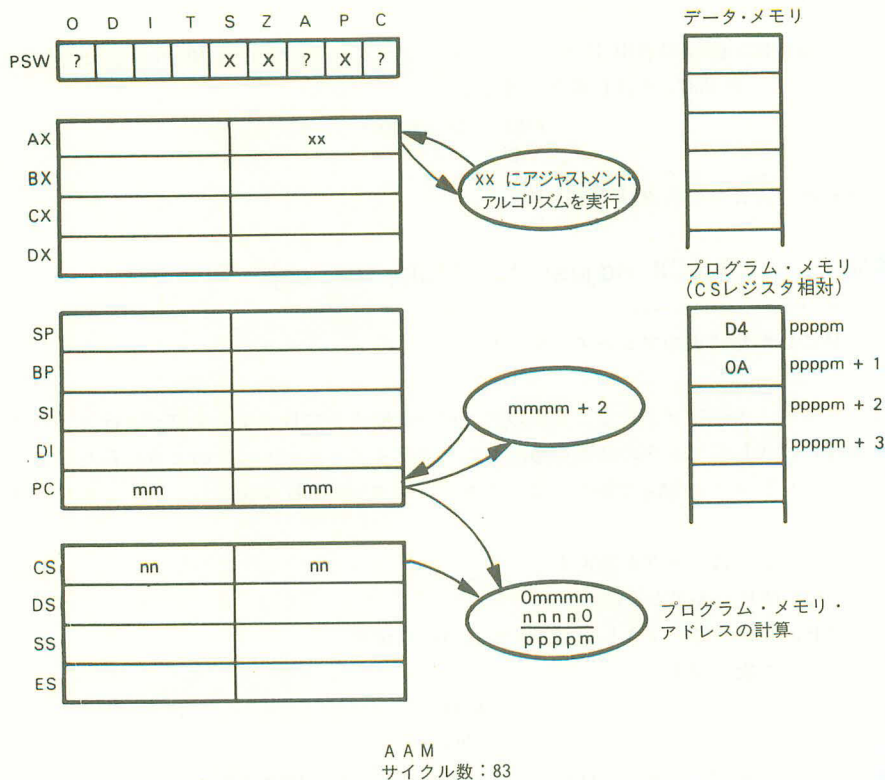
$$\underbrace{\text{AAM}}_{\text{D40A}}$$

ALレジスタが07_{16}を含み，BLレジスタが09_{16}を含むと仮定すると，

```
MUL     AL, BL
AAM
```

の実行後，AXレジスタは0603_{16}となる．MUL命令の結果，$3F_{16}$がALレジスタにストアされる．アジャストメント・アルゴリズムを行なうことによってAXレジスタは0603_{16}となり，フラグは次のように設定される．

CF，OF，AF：不定．
SF：ALレジスタの最上位ビットが0なので，SF＝0．
ZF：ALレジスタが0でないので，ZF＝0．
PF：ALレジスタに1のビットが2個あるので，PF＝1．

AAM
サイクル数：83

AAS　　　(ASCII Adjust for Subtraction)

ASCII減算結果のアジャストを行なう.

この命令は，オペランドとして2つのASCIIキャラクタによる減算が実行されたことを前提に，ALレジスタの結果を補正する．アジャストメントは次のように行なわれる．

1．ALレジスタの下位4ビットが0と9の間で，しかもAFフラグが0ならば，ステップ3へ.
2．ALレジスタの下位4ビットがAとFの間，あるいはAFフラグが1ならば，ALレジスタから6を引き，AHレジスタから1を減じて，AFフラグを1にセットする.
3．ALレジスタの上位4ビットをクリヤする.
4．CFフラグにAFフラグの値を設定する.

命令コードを次に示す.

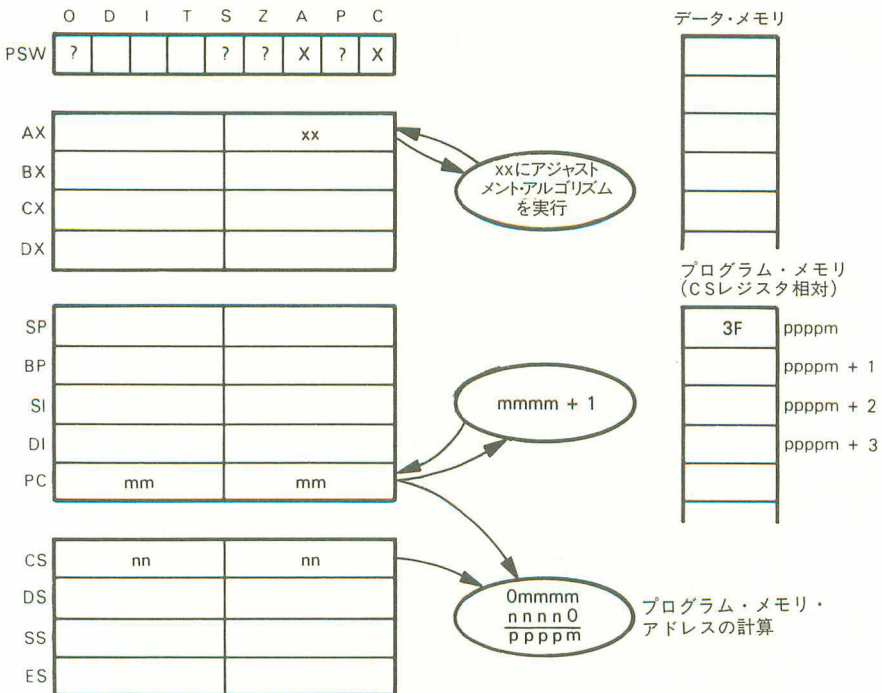
```
AAS
3F
```

たとえば，ＡＸレジスタが0438_{16}を含むと仮定する．

```
SUB    AL,35H
AAS
```

の実行後，ＡＸレジスタは0403_{16}となる．ＳＵＢ命令の結果，

$$
\begin{array}{rr}
38_{16} = & 0011\ \ 1000 \\
35_{16}\text{の2の補数} = & 1100\ \ 1011 \\
\hline
 & 0000\ \ 0011
\end{array}
$$

03_{16}がＡＬにストアされる．ＡＡＳ命令のアジャストメント・アルゴリズムの実行では，この場合ＡＸレジスタは変更されない．ＡＦとＣＦのフラグは０に設定される．

ＡＡＳ
サイクル数：４

注）

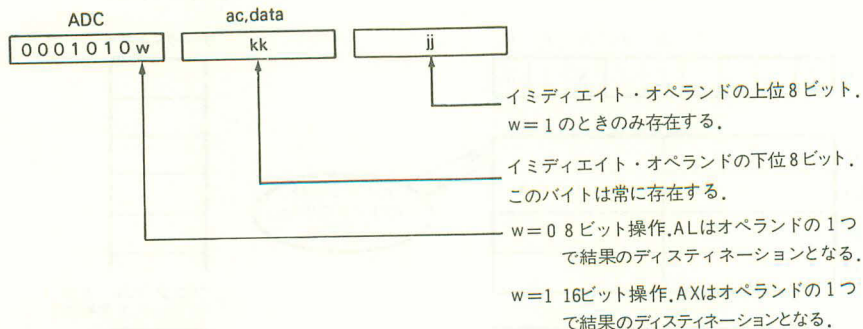
1．ASCII 加算結果のアジャストについては，ＡＡＡ命令参照．　パック形式ＢＣＤの加算と減算の結果のアジャストについては，ＤＡＡとＤＡＳの命令を参照．

2．この命令の実行により，ＰＦ，ＺＦ，ＳＦ，ＯＦの各フラグの値は不定となる．

ADC　　ac, data　　　　(Add with Carry)

ＡＸあるいはＡＬのレジスタに，キャリーと共にイミディエイト・データを加算する．

この命令は，後続のプログラム・メモリ・バイトに存在するイミディエイト・データとキャリーを，ＡＬ（８ビット操作）あるいはＡＸ（16ビット操作）のレジスタに加算するために用いられる．

命令コードを次に示す．

たとえば，ＡＸレジスタが$4F3D_{16}$を含み，キャリーが１の場合を考える．

ADC　　AX,0FD81H

の実行後，ＡＸレジスタは$4CBF_{16}$を含み，キャリーは１となる．

ADC AX, jjkk
サイクル数：4

注）

1．この命令は，8080の命令 ACI data と同じ機能を果たす．さらに，この命令はキャリーと共にイミディエイトの16ビットの加算を行なえる．

ADC mem/reg, data　　(Add with Carry)

レジスタあるいはメモリに対して，キャリーと共にイミディエイトを加算する．

この命令は，後続のプログラム・メモリ・バイトに存在するイミディエイト・データとキャリーを，指定されたレジスタあるいはメモリに加算するために用いられる．8ビットあるいは16ビットの操作が指定できる．

命令コードを次に示す．

ADC	mem/reg.	data	
100000sw	mod 010 r/m	kk	jj

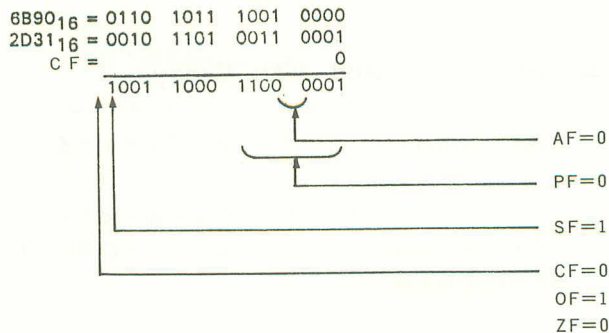
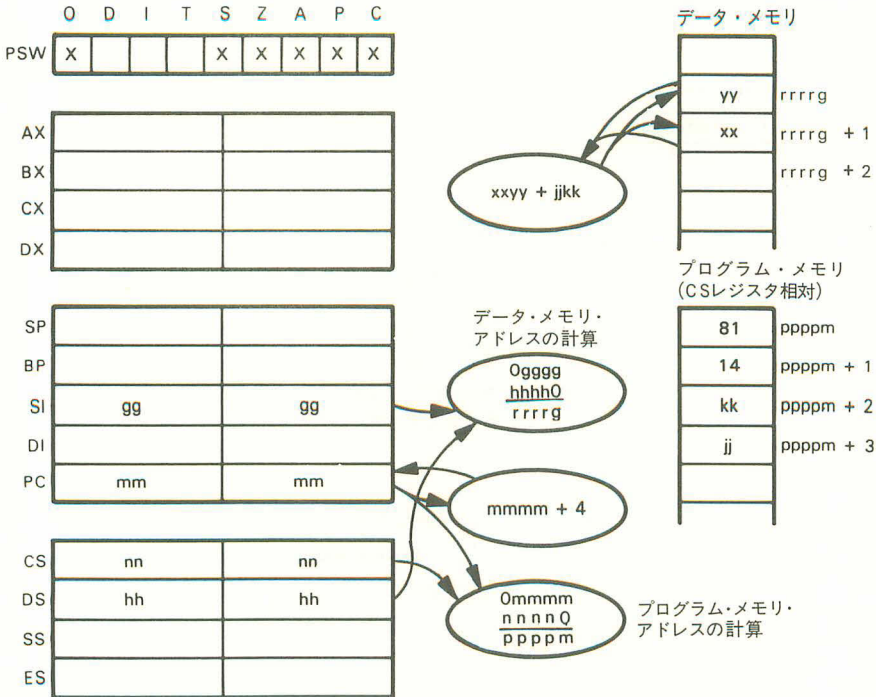
- jj：16ビット・イミディエイト・オペランドの上位バイト．s＝0でw＝1のときのみ存在する．
- kk：イミディエイト・オペランドの下位バイト．このバイトは常に存在する．
- mod 010 r/m：アドレッシング・モード・バイト．
- w：w＝0　8ビット操作　w＝1　16ビット操作
- s：sは符号拡張ビット
  - w＝0のときは無視される．
  - w=1のとき
    - s＝0ならばイミディエイト・オペランドの全16ビットが存在する．
    - s＝1ならばイミディエイト・オペランドの下位8ビットのみが存在する．16ビット・オペランドの上位8ビットはkkの上位ビットを符号拡張して構成される．

ＤＳレジスタがE400$_{16}$を含み，ＳＩレジスタが0040$_{16}$を含み，メモリ位置E4040$_{16}$のワードが6B90$_{16}$で，キャリーは0であると仮定する．

ADC　〔SI〕, 2D31H

の実行後，メモリ位置E4040$_{16}$のワードは98C1$_{16}$を含み，キャリーは0となる．

ADC 〔SI〕, jjkk

サイクル数：メモリに対して：17＋EA
　　　　　　レジスタに対して：4

注）

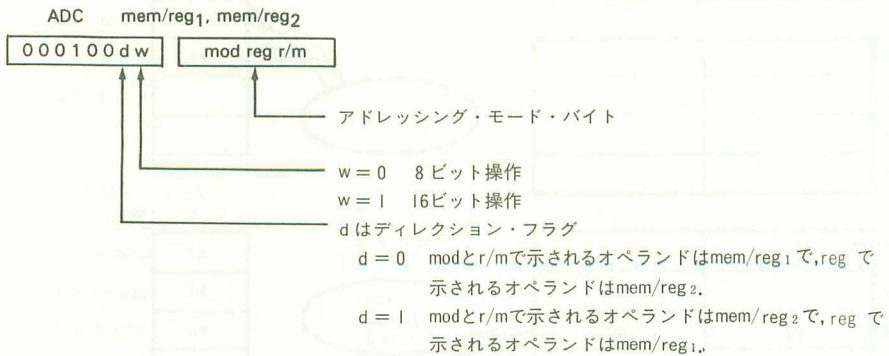
1．この命令は通常ALあるいはAXのレジスタに対しては用いられない．その目的のためには命令 ADC ac, data がある．

2．セグメント・レジスタは，この命令でオペランドとしては指定されない．

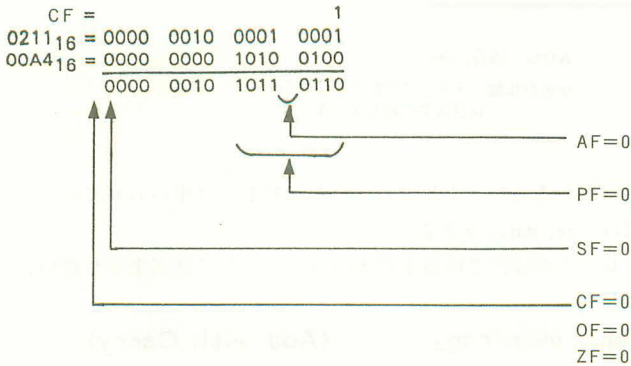
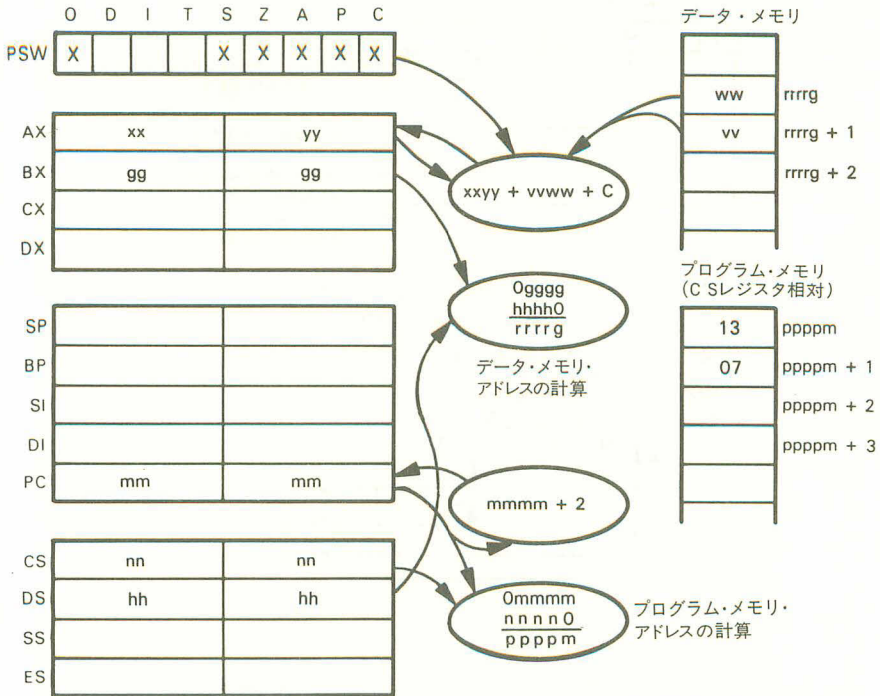
ADC　mem/reg₁, mem/reg₂　　(Add with Carry)

$\left\{\begin{array}{l}\text{レジスタからレジスタに}\\\text{レジスタからメモリに}\\\text{メモリからレジスタに}\end{array}\right\}$キャリーと共に加算を行なう．

mem/reg_2で指定されるレジスタあるいはメモリ位置の内容とキャリーとを，mem/reg_1で指定されるレジスタあるいはメモリ位置の内容に加算する．8あるいは16ビットの操作が指定できる．mem/reg_1あるいはmem/reg_2はメモリ・オペランドとなるが，オペランドの一方はレジスタ・オペランドでなければならない．

命令コードを次に示す.

AXレジスタが0211_{16}を含み，BXレジスタが0084_{16}を含み，DSレジスタが$1C00_{16}$を含み，キャリーが1で，$1C084_{16}$のメモリ・ワードの内容が$00A4_{16}$であると仮定する．

ADC AX,〔BX〕

の実行後，AXレジスタは$02B6_{16}$を含み，フラグは次のようになる．

ADC　AX,〔BX〕
サイクル数：メモリからレジスタに対して：９＋EA
　　　　　　レジスタからメモリに対して：16＋EA
　　　　　　レジスタからレジスタに対して：３

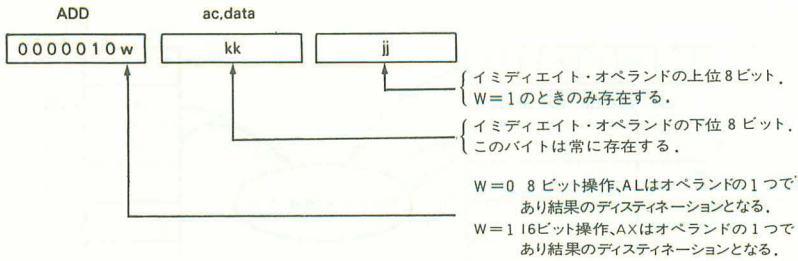
注）

1．この命令は通常ＡＸあるいはＡＬのレジスタに対しては用いられない．ＡＤＣ ac, data 命令はこの機能をより少ないバイト数で行なう．

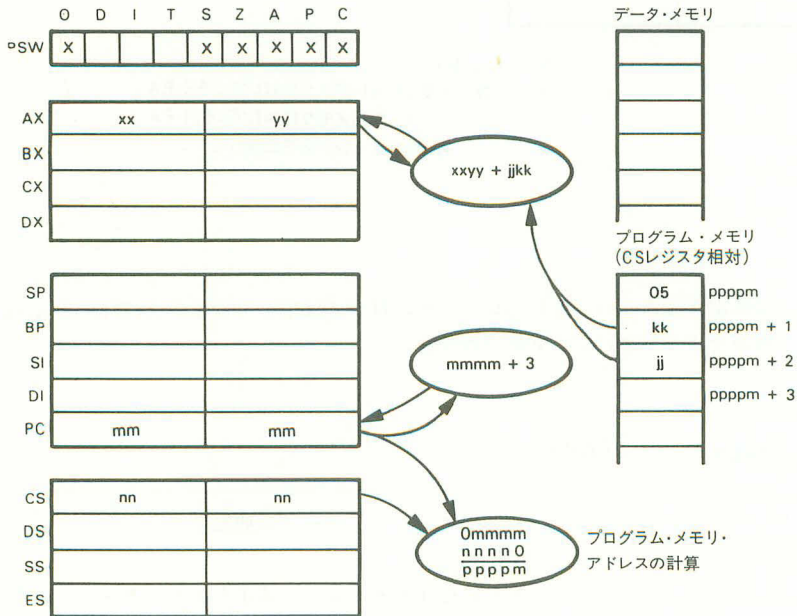
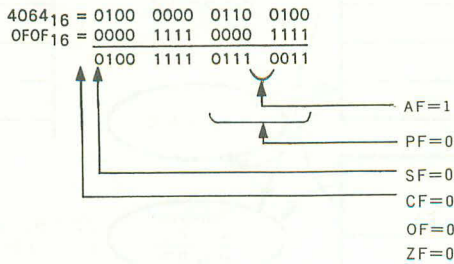
ADD　ac, data　　　　(Add)

ＡＸあるいはＡＬのレジスタにイミディエイト・データを加算する．

この命令は，後続のプログラム・メモリに存在するイミディエイト・データを，AL(8ビット操作）あるいはＡＸ（16ビット操作）のレジスタに加算するために用いられる．

命令コードを次に示す．

ＡＸレジスタが 4064_{16} を含み，キャリーが１であると仮定する．

ADD　　AX,0F0FH

の実行の結果，ＡＸレジスタは $4F73_{16}$ を含む．

ADD　AX, jjkk
サイクル数：4

注)

1．この命令は，8080の命令 ADI data と同じ機能を果たす．この命令はさらに16ビット・イミディエイト・データの要素を加算することができる．

ADD mem/reg, data　　　(Add)

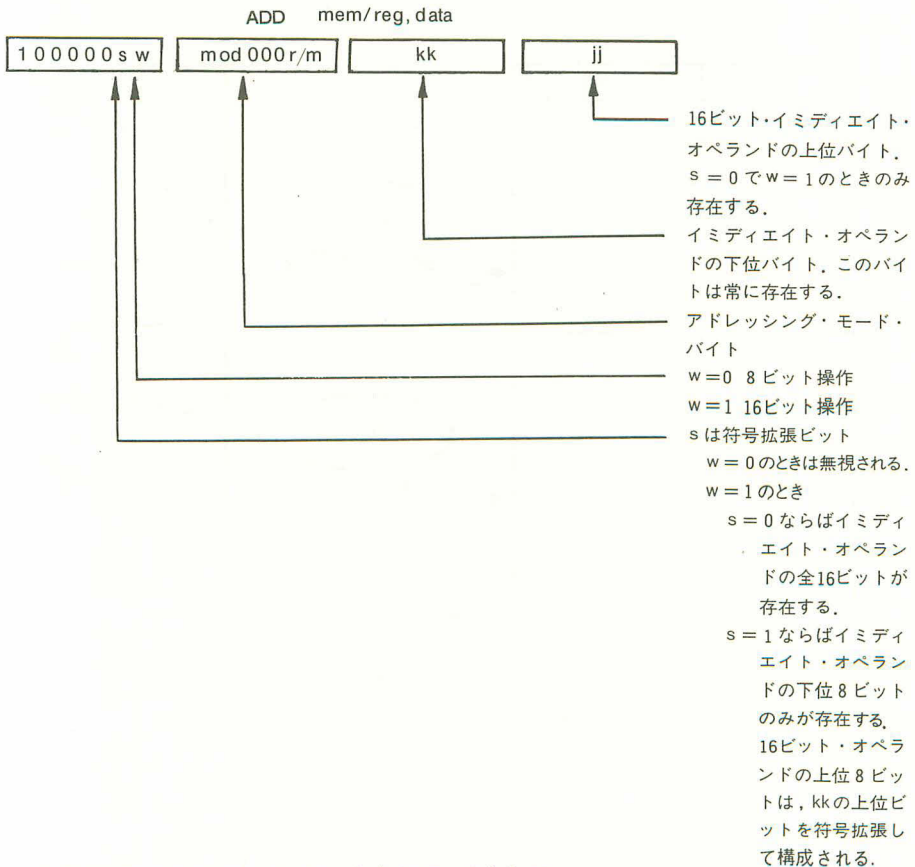
レジスタあるいはメモリにイミディエイト・データを加算する．

この命令は，後続のプログラム・メモリ・バイトに存在するイミディエイト・データを，指定されたレジスタあるいはメモリ位置に加算するために用いられる．8あるいは16ビットの操作が指定できる．

命令コードを次に示す．

ADD　mem/reg, data

100000sw	mod000r/m	kk	jj

- jj：16ビット・イミディエイト・オペランドの上位バイト．s＝0でw＝1のときのみ存在する．
- kk：イミディエイト・オペランドの下位バイト．このバイトは常に存在する．
- mod000r/m：アドレッシング・モード・バイト
- w＝0　8ビット操作
  w＝1　16ビット操作
- sは符号拡張ビット
  w＝0のときは無視される．
  w＝1のとき
  s＝0ならばイミディエイト・オペランドの全16ビットが存在する．
  s＝1ならばイミディエイト・オペランドの下位8ビットのみが存在する．16ビット・オペランドの上位8ビットは，kkの上位ビットを符号拡張して構成される．

たとえば，DXレジスタが4652_{16}を含んでいるとして，

```
ADD	DX,0F0F0H
```

を実行すると，ＤＸレジスタの内容は 3742_{16} に変わる．

O D I T S Z A P C
PSW X X X X X X
AX
BX
CX
DX xx yy
xxyy + jjkk
データ・メモリ
プログラム・メモリ
(CSレジスタ相対)
81 ppppm
C2 ppppm + 1
kk ppppm + 2
jj ppppm + 3
SP
BP
SI
DI
PC mm mm
mmmm + 4
CS nn nn
DS
SS
ES
0mmmm
nnnn0
ppppm
プログラム・メモリ・
アドレスの計算

ADD　DX，jjkk
サイクル数：メモリに対して：17＋EA
　　　　　　レジスタに対して：4

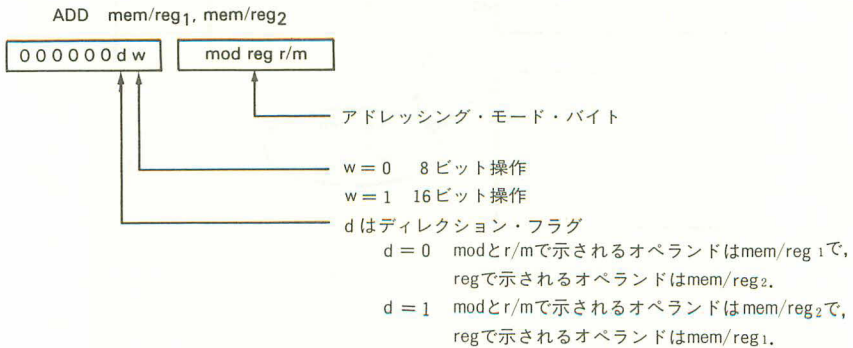
注）

1．この命令は通常ＡＸあるいはＡＬのレジスタに対しては用いられない．ＡＤＤ ac, data 命令はこの機能をより少ないバイト数で行なう．

ADD mem/reg_1 , mem/reg_2　　(Add)

レジスタをレジスタに
レジスタをメモリに　　｝加算する．
メモリをレジスタに

mem/reg_2 で指定されるレジスタあるいはメモリ位置の内容を，mem/reg_1 で指定されるレジスタあるいはメモリ位置の内容に加える．8あるいは16ビットの操作が指定できる．mem/reg_1 あるいは mem/reg_2 はメモリ・オペランドとなるが，オペランドの一方はレジスタ・オペランドでなければならない．

命令コードを次に示す．

CXレジスタが 0029_{16} を含み，SIレジスタの内容が $04ED_{16}$ であると仮定する．

ADD　SI ,CX

の実行後，SIレジスタの内容とフラグは次のように変化する．

O D I T S Z A P C

PSW X X X X X X

データ・メモリ

AX
BX
CX vv ww
DX

vvww + xxyy

プログラム・メモリ
(CS レジスタ相対)

SP
BP
SI xx yy
DI
PC mm mm

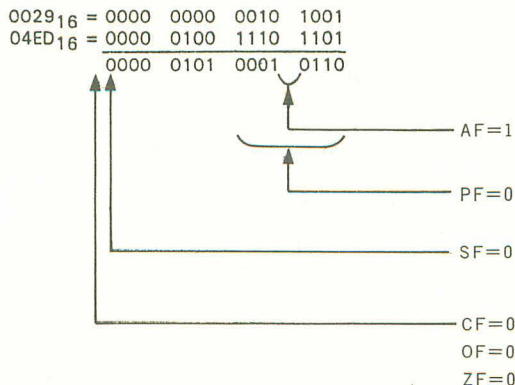
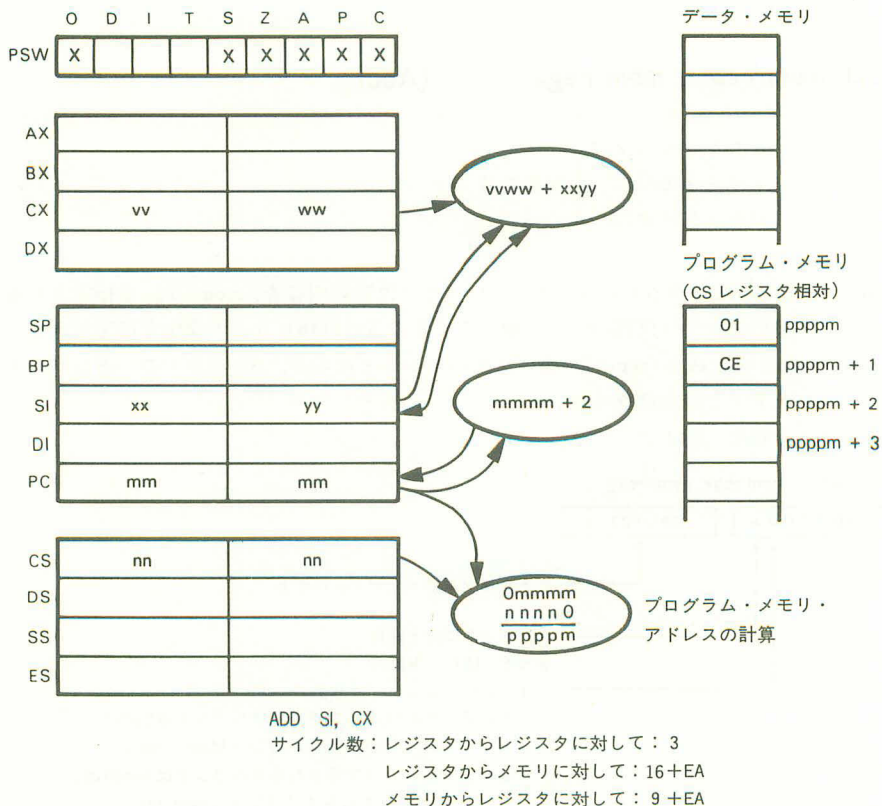
01 ppppm
CE ppppm + 1
ppppm + 2
ppppm + 3

mmmm + 2

CS nn nn
DS
SS
ES

0mmmm
nnnn0
ppppm

プログラム・メモリ・
アドレスの計算

ADD SI, CX
サイクル数：レジスタからレジスタに対して：3
レジスタからメモリに対して：16＋EA
メモリからレジスタに対して：9＋EA

AND ac, data (AND)

ＡＬあるいはＡＸレジスタとイミディエイト・データのＡＮＤをとる．

この命令は，後続のプログラム・メモリ・バイトに存在するイミディエイト・データと，ＡＬ(8ビット操作) あるいはＡＸ(16ビット操作) のレジスタの内容のＡＮＤを行なうために用いられる．

命令コードを次に示す．

例として，ALレジスタが$C3_{16}$を含む場合を考える．

AND　AL,7FH.

の実行後，ALレジスタは43_{16}を含む．

O D I T S Z A P C

PSW 0 X X ? X 0

AX xx
BX
CX
DX

xx ∧ kk

データ・メモリ

プログラム・メモリ・
(CSレジスタ相対)

24 ppppm
kk ppppm+1
ppppm+2
ppppm+3

SP
BP
SI
DI
PC mm mm

mmmm+2

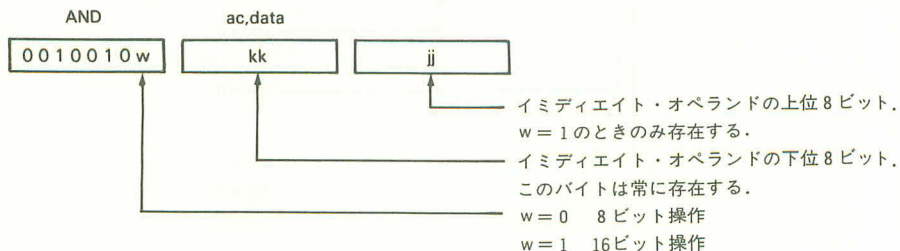
CS nn nn
DS
SS
ES

0mmmm
nnnn0
ppppm

プログラム・メモリ・
アドレスの計算

AND　AL, kk
サイクル数：4

注）

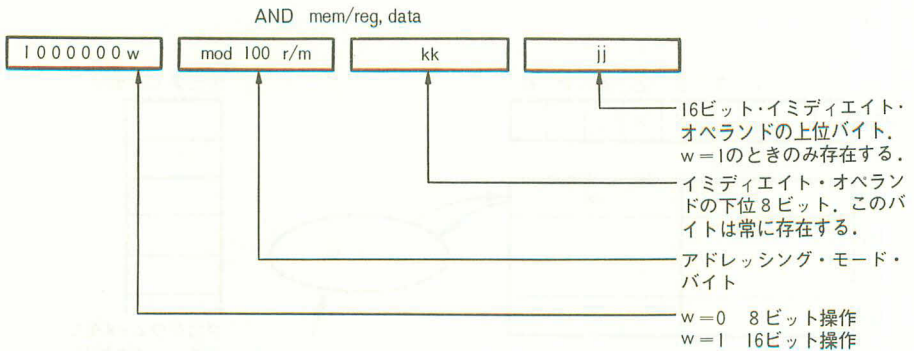
1．この命令は，8080の命令　ANI data と同じ機能を果たす．しかし，16ビット操作も

可能である．

2．他の汎用レジスタあるいはメモリとイミディエイトのＡＮＤが必要ならば，ＡＮＤ mem/reg,data 命令を参照．

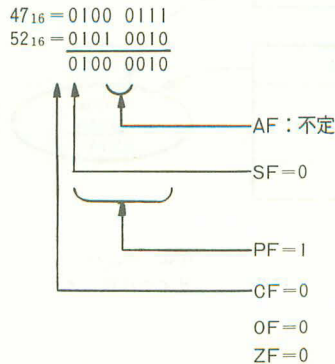
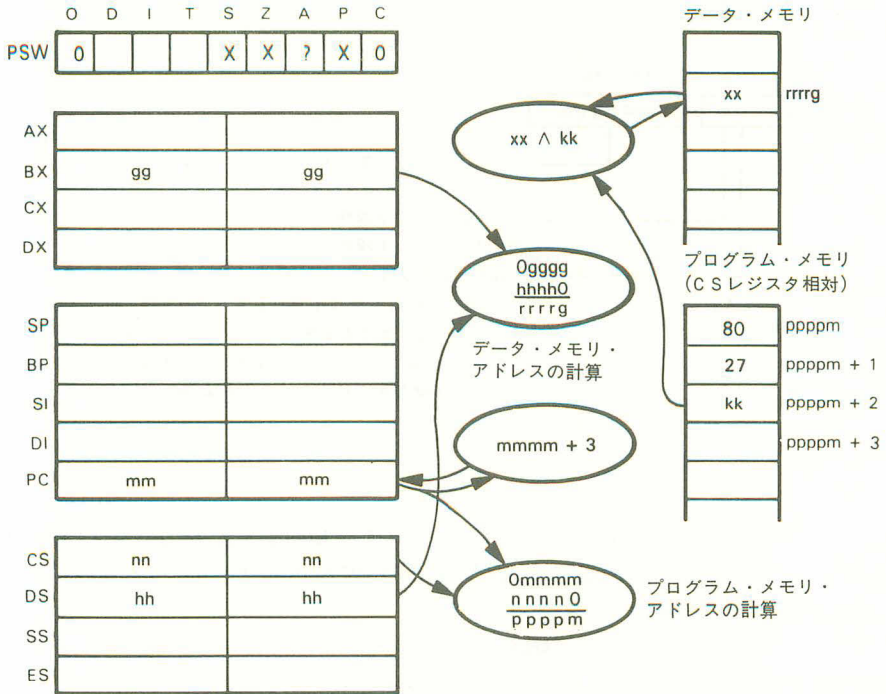
AND mem/reg, data (AND)

レジスタあるいはメモリとイミディエイト・データのＡＮＤをとる．

後続のプログラム・メモリ・バイトに存在するイミディエイト・データと，指定されたレジスタあるいはメモリ位置のＡＮＤをとる．8あるいは16ビットの操作が指定できる．

命令コードを次に示す．

ＢＸレジスタが 0104_{16} を含み，ＤＳレジスタが 0000_{16} を含み，メモリ位置 00104_{16} のバイトが 47_{16} の場合を考える．

AND 〔BX〕,52H

の実行後，メモリ位置 00104_{16} は 42_{16} を含む．

AND 〔BX〕, kk
サイクル数：メモリに対して：17＋EA
　　　　　　レジスタに対して：4

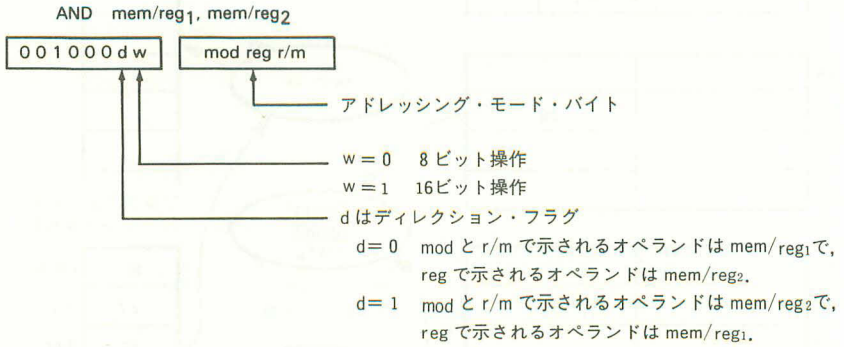
注)

1．この命令は通常ＡＸあるいはＡＬのレジスタに対しては用いられない．この機能に対しては，命令　ＡＮＤ　ac,data　が用いられる．

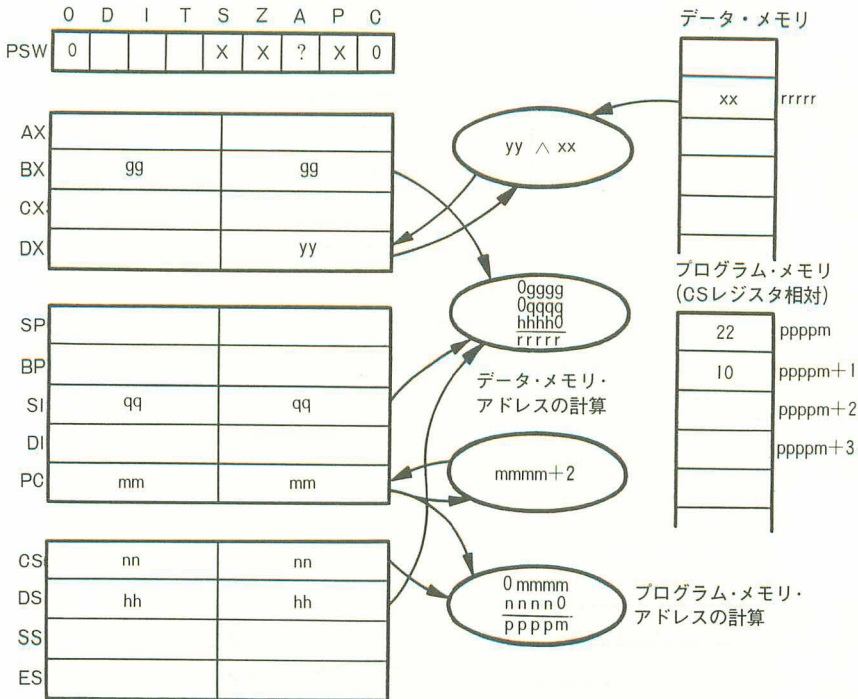
AND　mem/reg₁，mem/reg₂　　　　(AND)

$\left\{\begin{array}{l}\text{レジスタとレジスタ}\\\text{レジスタとメモリ}\\\text{メモリとレジスタ}\end{array}\right\}$ でＡＮＤをとる．

mem/reg_2 で指定されるレジスタあるいはメモリ位置の内容と，mem/reg_1 で指定されるレジスタあるいはメモリ位置の内容のＡＮＤをとる．８あるいは16ビットの操作が指定できる．mem/reg_1 あるいは mem/reg_2 はメモリ・オペランドとなるが，オペランドの一方はレジスタ・オペランドでなければならない．

命令コードを次に示す.

例として, DLレジスタが06_{16}を含み, DSレジスタが$B000_{16}$を含み, BXレジスタが0010_{16}を含み, SIレジスタが0006_{16}を含み, メモリ位置$B0016_{16}$のバイトが$F1_{16}$を含む場合を考える.

AND DL,〔BX＋SI〕

の実行後, DLレジスタは00_{16}を含み, フラグは次のようになる.

AND　DL,〔BX+SI〕

サイクル数：メモリからレジスタに対して：9＋EA

　　　　　　レジスタからメモリに対して：16+EA

　　　　　　レジスタからレジスタに対して：3

CALL　addr　　　　(CALL)

オペランドで指定されるサブルーチンをＣＡＬＬする（セグメント間）.

ＣＳとＰＣレジスタの内容をスタックのトップにストアする，すなわち，ＣＡＬＬに続く命令のアドレスをスタックのトップにプッシュする．後続の４個のメモリ・バイトの内容をＰＣとＣＳのレジスタに設定する．このバイトは次のように設定される．

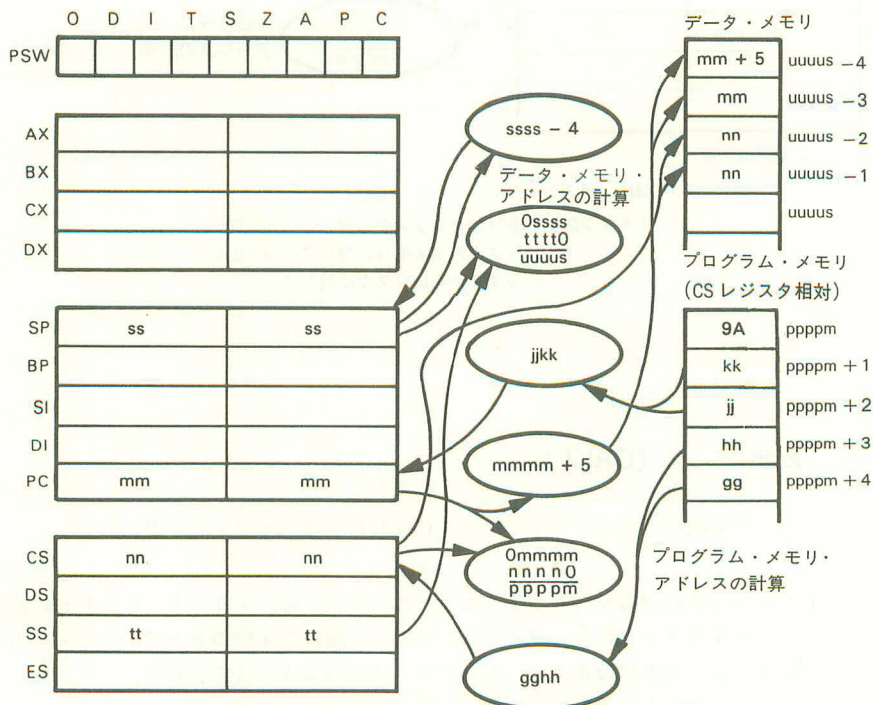
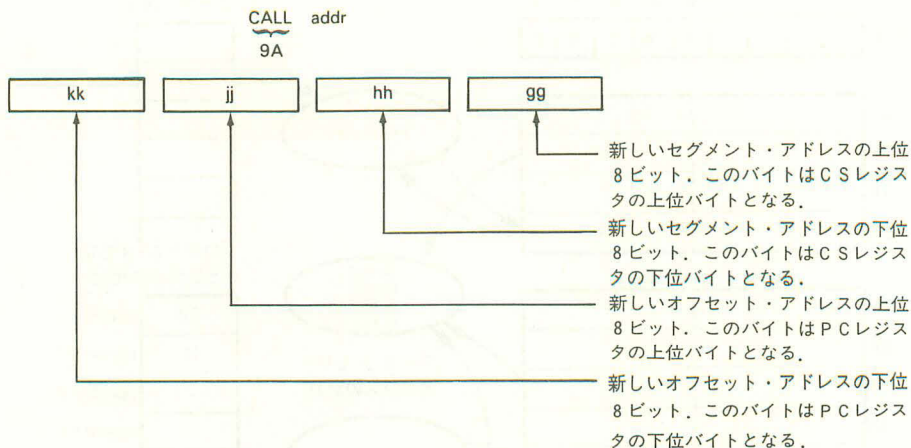
1．この命令の２番目と３番目のバイトをＰＣレジスタにストアする．

2．この命令の４番目と５番目のバイトをＣＳレジスタにストアする．

命令コードを次に示す．

O D I T S Z A P C
PSW
AX
BX
CX
DX
SP ss ss
BP
SI
DI
PC mm mm
CS nn nn
DS
SS tt tt
ES
ssss − 4
データ・メモリ・アドレスの計算
0ssss
tttt0
uuuus
jjkk
mmmm + 5
0mmmm
nnnn0
ppppm
gghh
データ・メモリ
mm + 5 uuuus −4
mm uuuus −3
nn uuuus −2
nn uuuus −1
uuuus
プログラム・メモリ
(CSレジスタ相対)
9A ppppm
kk ppppm + 1
jj ppppm + 2
hh ppppm + 3
gg ppppm + 4
プログラム・メモリ・アドレスの計算

CALL addr
サイクル数：28

注)

1．ＣＡＬＬには次の４つのタイプがある．

　　ＣＡＬＬ　addr：セグメント間ＣＡＬＬ

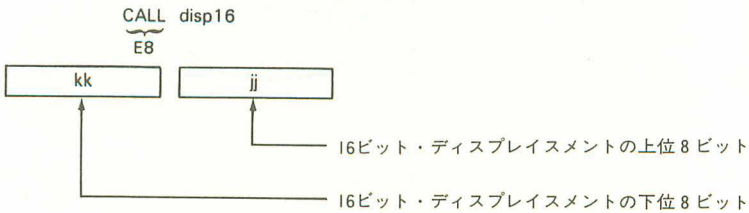
　　ＣＡＬＬ　mem：セグメント間インダイレクトＣＡＬＬ

　　ＣＡＬＬ　disp：セグメント内ＣＡＬＬ

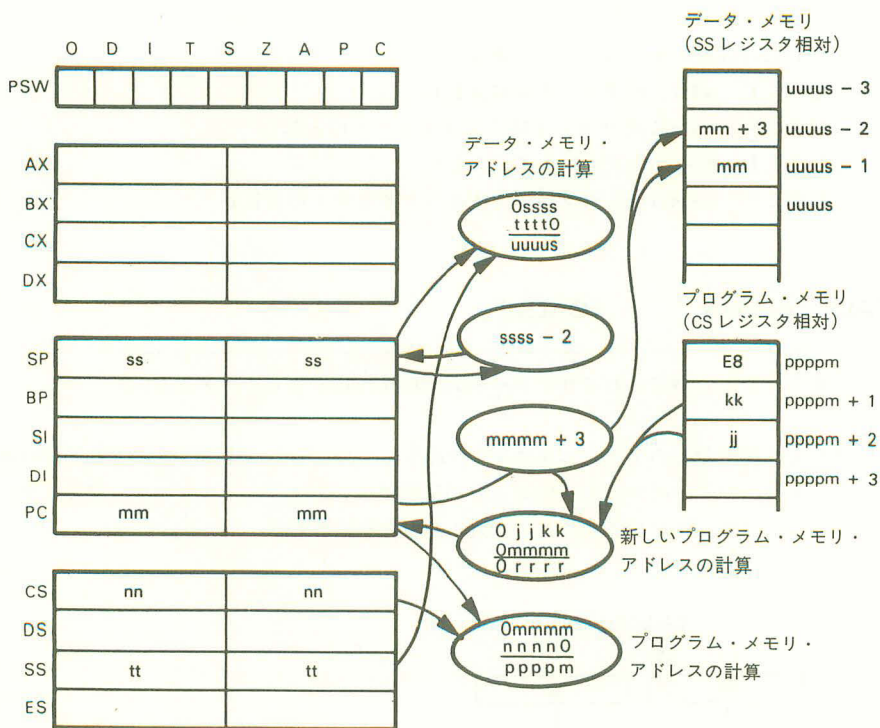
　　ＣＡＬＬ　mem/reg：セグメント内インダイレクトＣＡＬＬ

CALL disp16　　(CALL)

オペランドで指定されるサブルーチンをＣＡＬＬする（セグメント内）．

ＣＡＬＬに続く命令のアドレスをスタックのトップにプッシュする．次の２つのプログラム・メモリ・バイトの内容を，16ビットの符号なしディスプレイスメントと見なして，プログラム・カウンタに加える．この位置から実行を続ける．

例として，次の一連の命令を考える．

```
        CALL   SUBR
        AND    AL,7FH
         -
         -
         -
SUBR    PUSH   AX
```

ＣＡＬＬ命令実行後，ＡＮＤ命令のアドレスはスタックにプッシュされ，ＳＵＢＲのＰＵＳＨ命令が次に実行される．

CALL　jjkk
サイクル数：19

注)

1．ＣＡＬＬには次の4つのタイプがある．

　ＣＡＬＬ　disp：セグメント内ＣＡＬＬ

　ＣＡＬＬ　mem/reg：セグメント内インダイレクトＣＡＬＬ

　ＣＡＬＬ　addr：セグメント間ＣＡＬＬ

　ＣＡＬＬ　mem：セグメント間インダイレクトＣＡＬＬ

CALL　mem　　　(CALL)

オペランドで指定されるサブルーチンをＣＡＬＬする（セグメント間）．

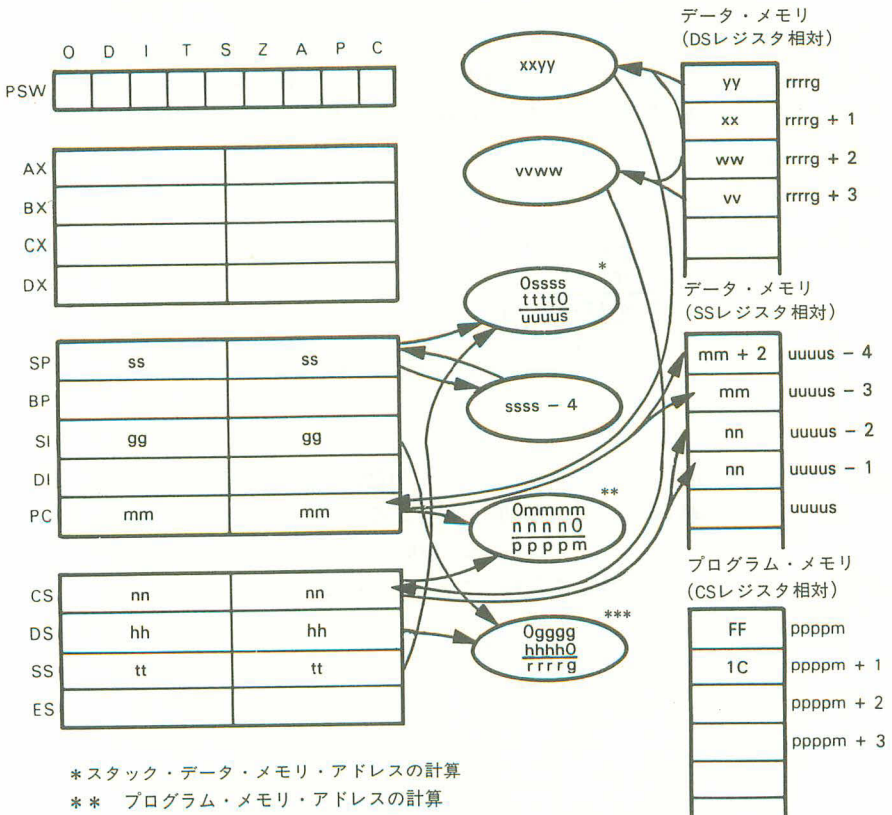
ＣＳとＰＣのレジスタの内容をスタックのトップにストアする．すなわち，ＣＡＬＬに続く命令のアドレスをスタックにプッシュする．指定されたメモリ位置のワードをＰＣレジスタに移し，それに続くワードをＣＳレジスタに移す．この位置から実行を続ける．

命令コードを次に示す．

DSレジスタが0400_{16}を含み，SIレジスタが0004_{16}を含み，04004_{16}のメモリ・ワードが0100_{16}で04006_{16}のメモリ・ワードが$0FE0_{16}$であると仮定する．

CALL 〔SI〕

の実行後，PCレジスタは0100_{16}を含み，CSレジスタは$0FE0_{16}$を含む．実行は$0FF00_{16}$から続けられる．

＊スタック・データ・メモリ・アドレスの計算
＊＊　プログラム・メモリ・アドレスの計算
＊＊＊　データ・メモリ・アドレスの計算

CALL [SI]
サイクル数：37 + EA

注)

1．ＣＡＬＬには次の４つのタイプがある．

　　ＣＡＬＬ　mem：セグメント間インダイレクトＣＡＬＬ

　　ＣＡＬＬ　addr：セグメント間ＣＡＬＬ

　　ＣＡＬＬ　mem/reg：セグメント内インダイレクトＣＡＬＬ

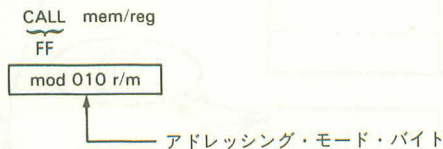
　　ＣＡＬＬ　disp：セグメント内ＣＡＬＬ

2．mod＝11のとき，この操作は定義されない．

CALL mem/reg (CALL)

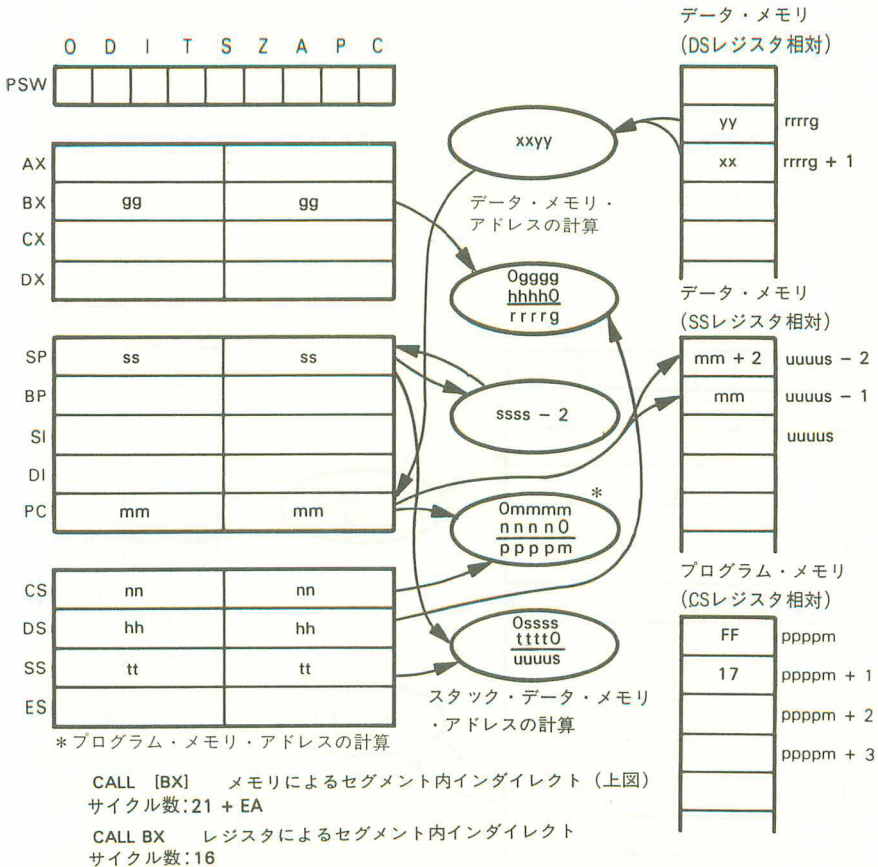
オペランドで指定されるサブルーチンをＣＡＬＬする（セグメント内）．

ＣＡＬＬに続く命令のアドレスをスタックのトップに格納する．指定されたオペランドがレジスタならば，そのレジスタの内容をＰＣレジスタに移動する．指定されたオペランドがメモリ位置ならば，指定されたメモリ位置の内容をＰＣレジスタに移動する．この位置から実行を続ける．

命令コードを次に示す．

ＰＣレジスタがＦＦ00_{16}を含み，ＤＳレジスタが0100_{16}を含み，ＢＸレジスタが0026_{16}を含み，メモリ位置01026_{16}のワードが0240_{16}の場合を考える．

CALL 〔BX〕

の実行後，ＰＣレジスタは0240_{16}を含む．この位置から実行が続けられる．

注）

1．CALLには次の4つのタイプがある．

CALL　mem/reg：セグメント内インダイレクトCALL

CALL　disp：セグメント内CALL

CALL　mem：セグメント間インダイレクトCALL

CALL　addr：セグメント間CALL

CBW　(Convert Byte to Word)

ALレジスタの符号をAHレジスタに拡張する．

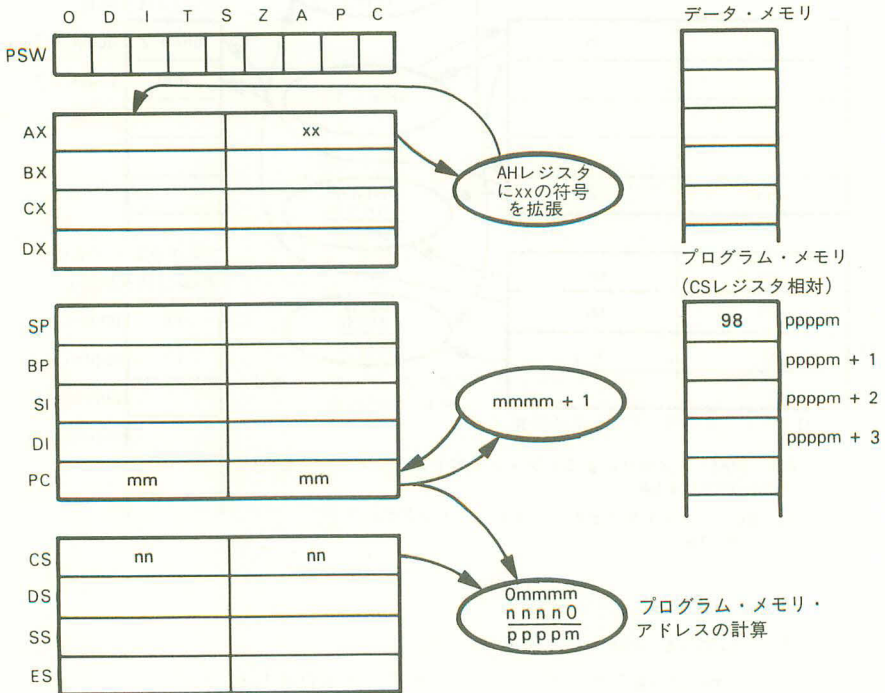
ALレジスタの最上位ビットが1ならばFF$_{16}$をAHレジスタにストアし，そうでなければ00$_{16}$をAHレジスタにストアする．

命令コードを次に示す.

$$\underbrace{\text{CBW}}_{98}$$

例として，ALレジスタが$4F_{16}$を含むならば，

CBW

の実行によって00_{16}がAHレジスタにストアされる.

CBW
サイクル数：2

注）

1．ステータスは影響を受けない.

2．ALレジスタの値は，＋127から－128の間の数を表わす，すなわち，ALは符号付き8ビットの値を含んでいなければならない.

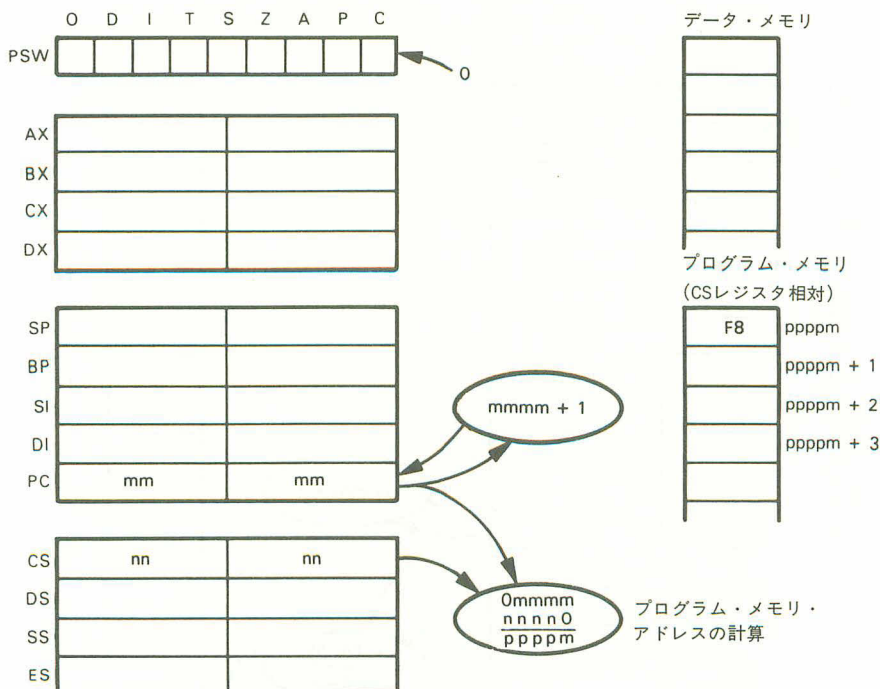
3．この命令は，16ビットのIMULあるいはIDIVの命令の前に，ALレジスタの拡張に使用できる.

CLC　　　(Clear Carry flag)

　　キャリー・フラグをクリヤする．

　この命令は，ＣＦフラグを０に設定する．他のステータスあるいはレジスタは影響を受けない．

　命令コードを次に示す．

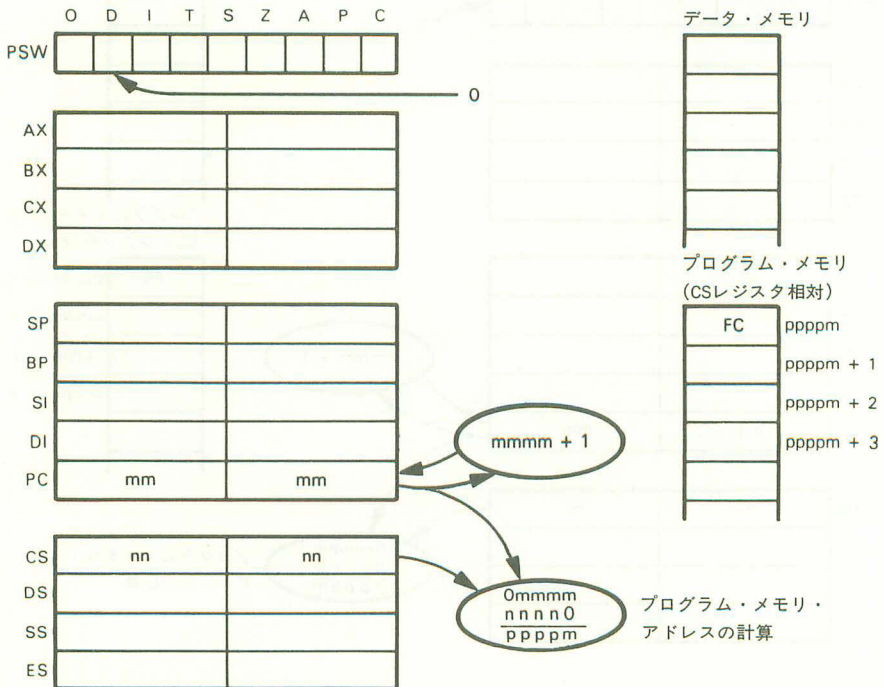
CLC
F8

CLC
サイクル数：2

CLD (Clear Direction flag)

ディレクション・フラグをクリヤする．

この命令は，ＤＦフラグを０に設定する．これは，ストリング操作で用いられているポインタを，その操作において自動的に増加させる効果をもっている．他のステータスあるいはレジスタは影響を受けない．

命令コードを次に示す．

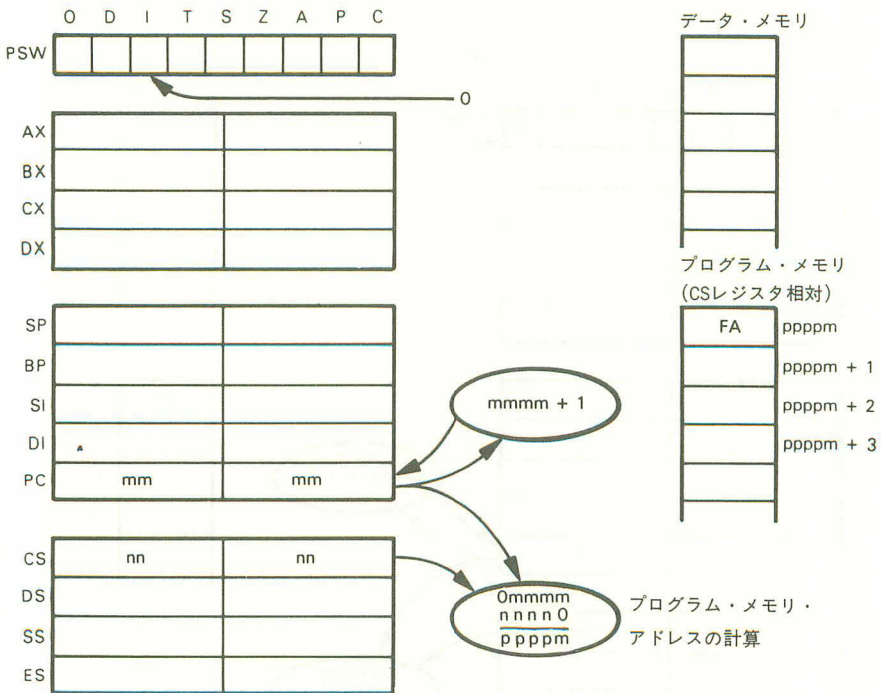
CLD
FC

CLD
サイクル数：2

CLI　　(Clear Interrupt flag)

インタラプト・フラグをクリヤする.

IFフラグを0に設定する. これは, NMIラインで生じるノンマスカブル・インタラプトを除くすべてのインタラプトを無効にする効果をもっている.

命令コードを次に示す.

CLI
FA

CLI
サイクル数 : 2

注)

1. この命令は, 8080の命令 DI と同じ機能を果たす.
2. 8086がインタラプト・リクエストを受け付けると, インタラプトは自動的に無効となることに注意.

CMC　　(Complement Carry flag)

キャリー・フラグのコンプリメントをとる．

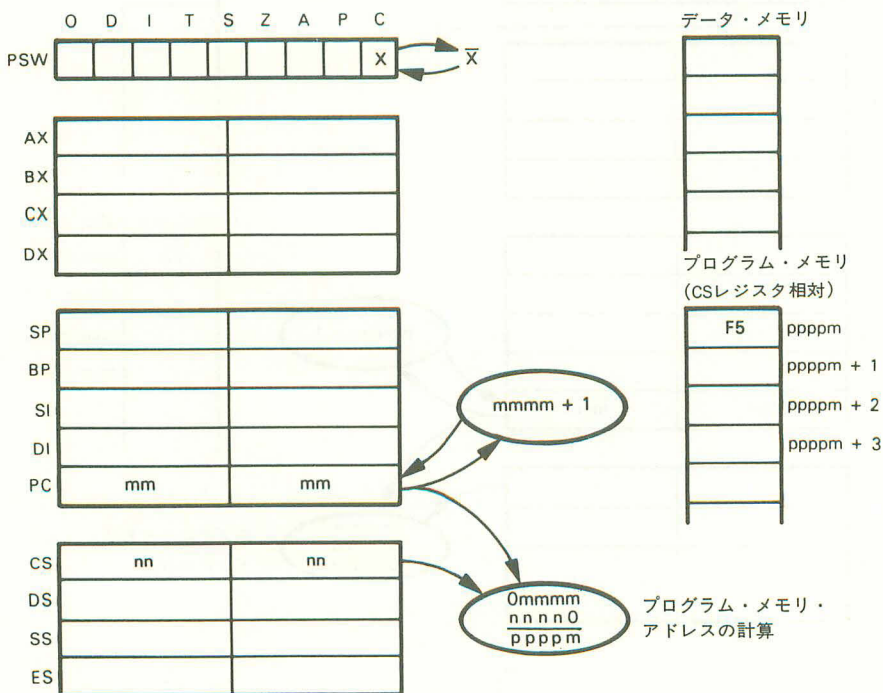
キャリー・フラグのコンプリメント（反転）を行なう．他のステータスあるいはレジスタは影響を受けない．

命令コードを次に示す．

CMC
F5

例えば，キャリー・フラグが0のとき，

CMC

を実行すると，キャリー・フラグは1となる．

CMC
サイクル数：2

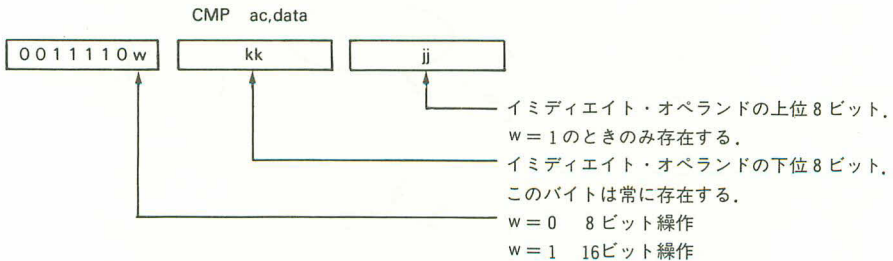
注）

1，この命令は，8080の命令 CMC と同じ機能を果たす．

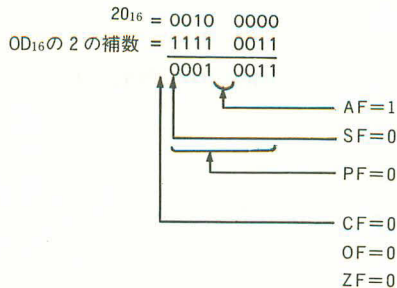
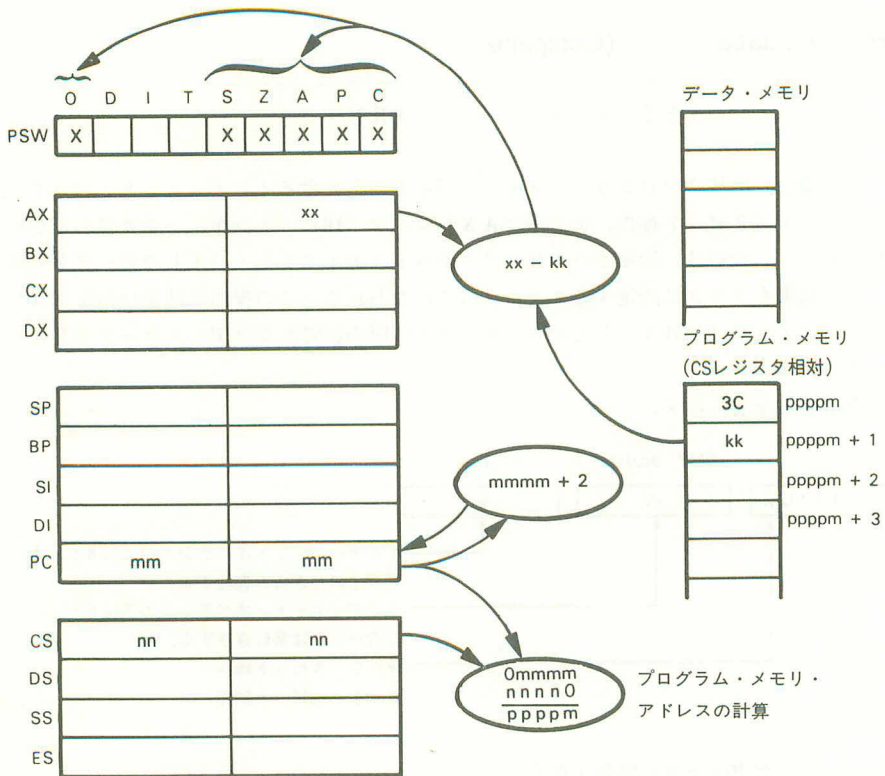
CMP　ac, data　　　　(Compare)

アキュムレータとイミディエイト・データを比較する.

この命令は，後続のプログラム・メモリ・バイトに存在するイミディエイト・データと，ALレジスタ（8ビット操作）あるいはAXレジスタ（16ビット操作）とを比較するために用いられる．比較は，指定されたレジスタからイミディエイト・バイトのデータを減算し，その結果をフラグに設定することによって行なわれる．この操作の結果は指定されたレジスタにストアはされず，したがってレジスタは何の影響も受けず，ステータスだけが変化する．

命令コードを次に示す．

ALレジスタが20_{16}を含む場合を考える．

CMP AL,0DH

の実行後，ALレジスタの値は20_{16}のままであるが，ステータスは次のように変更される．

CMP　AL,kk
サイクル数：4

注)

1．イミディエイト・データと，他の汎用レジスタあるいはメモリの内容との比較が必要な場合は，ＣＭＰ mem/reg, data 命令を参照．
2．この命令は，8080の命令 ＣＰＩ data と同じ機能を果たす．さらに，16ビットの比較も可能．

CMP mem/reg, data　　(Compare)

レジスタあるいはメモリとイミディエイト・データを比較する．

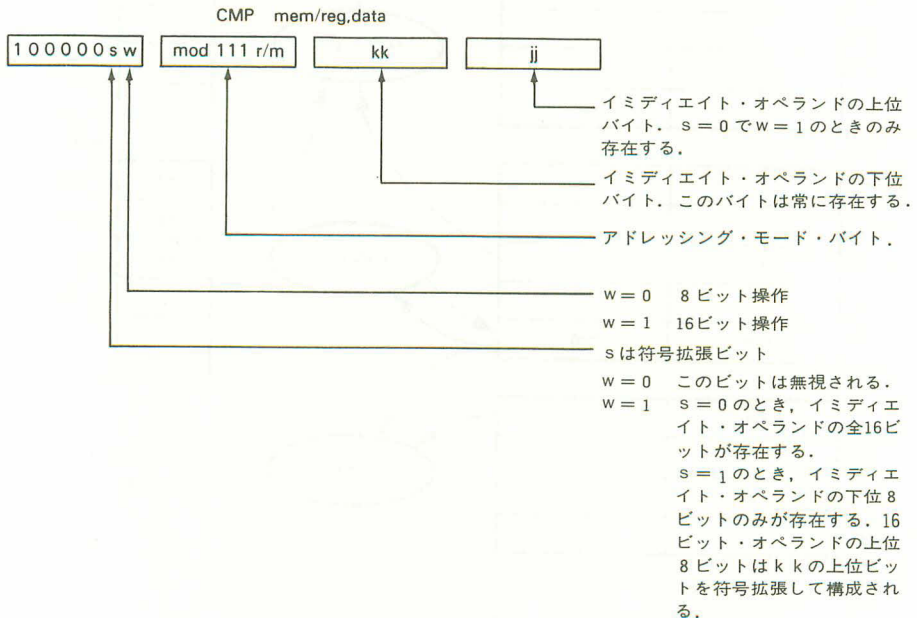
この命令は，後続のプログラム・メモリ・バイトに存在するイミディエイト・データと，指定されたレジスタあるいはメモリ位置との比較を行なう．比較は，指定されたメモリ位

置あるいはレジスタからイミディエイト・バイトのデータを減算し，その結果をフラグに設定することによって行なわれる．この操作の結果は指定されたレジスタあるいはメモリ位置には格納されず，したがって，レジスタあるいはメモリは何の影響も受けず，ステータスだけが変化する．8ビットあるいは16ビットの操作が指定できる．

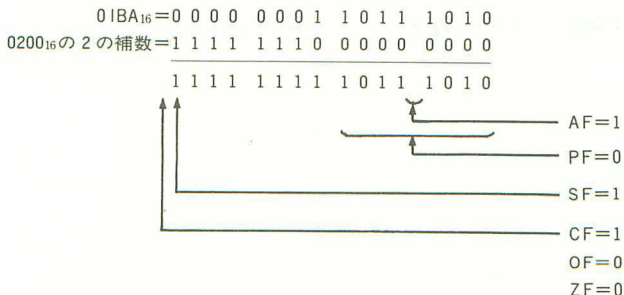
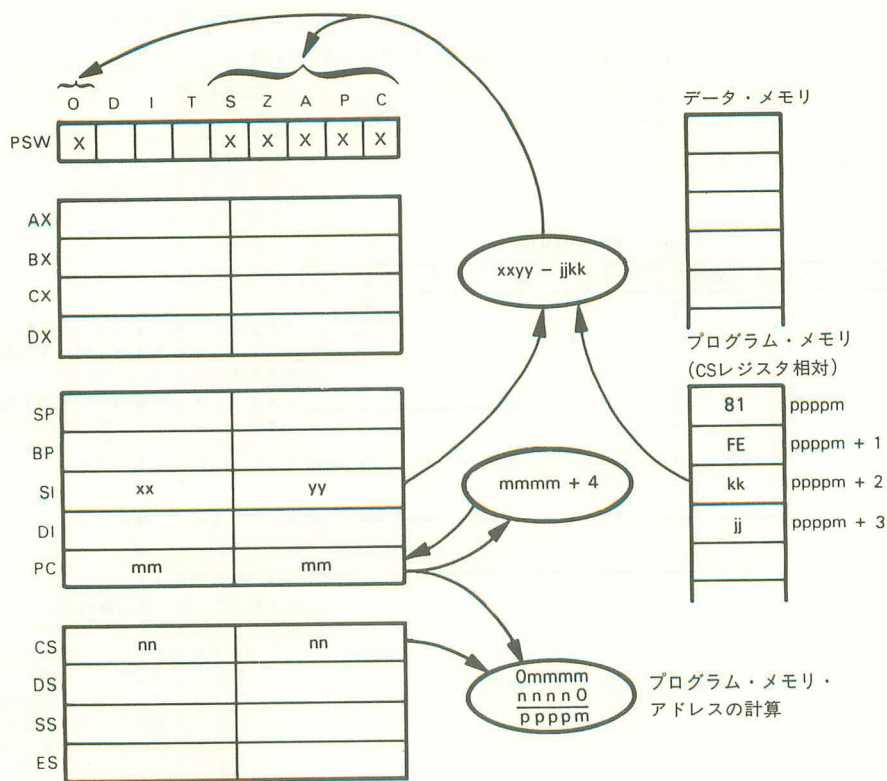
命令コードを次に示す．

CMP　mem/reg,data

100000sw	mod 111 r/m	kk	jj

- jj：イミディエイト・オペランドの上位バイト．s＝0でw＝1のときのみ存在する．
- kk：イミディエイト・オペランドの下位バイト．このバイトは常に存在する．
- mod 111 r/m：アドレッシング・モード・バイト．
- w：w＝0　8ビット操作
  w＝1　16ビット操作
- s：sは符号拡張ビット
  w＝0　このビットは無視される．
  w＝1　s＝0のとき，イミディエイト・オペランドの全16ビットが存在する．
  s＝1のとき，イミディエイト・オペランドの下位8ビットのみが存在する．16ビット・オペランドの上位8ビットはkkの上位ビットを符号拡張して構成される．

SIレジスタが$01AB_{16}$を含むと仮定する．

CMP　SI,0200H

の実行後，SIレジスタの値は$01BA_{16}$のままであるが，ステータスは次のように変更される．

CMP　SI, jjkk
サイクル数：レジスタ・オペランド：4
　　　　　　メモリ・オペランド：10＋EA

注）

1．この命令は，普通ＡＸあるいはＡＬのレジスタとイミディエイト・データを比較するためには用いられない．この目的のためには，命令 CMP ac, data がある．

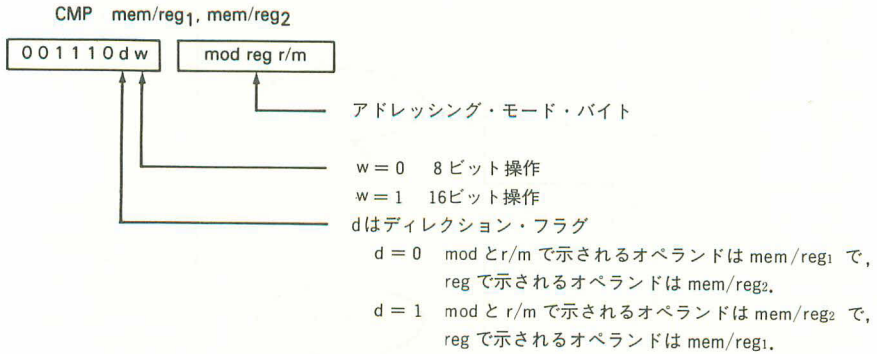
CMP　mem/reg_1, mem/reg_2　　(Compare)

{ レジスタとレジスタ / レジスタとメモリ / メモリとレジスタ } の比較を行なう．

mem/reg_2で指定されるレジスタあるいはメモリのオペランドのデータと，mem/reg_1で指定されるレジスタあるいはメモリのオペランドのデータとの比較を行なう．比較はmem/reg_1で指定されるデータからmem/reg_2で指定されるデータを減算し，その結果をフラグ

に設定することによって行なわれる．

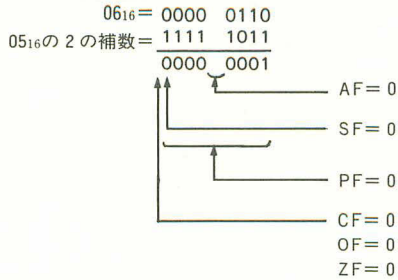
この操作によって，mem/reg_1 あるいは mem/reg_2 のどちらも影響を受けない．8あるいは16ビットの操作が指定できる．

命令コードを次に示す．

ＤＨレジスタが05_{16}を含み，ＣＬレジスタが06_{16}を含むと仮定する．

CMP　CL,DH

の実行後，ＣＬあるいはＤＨのレジスタのどちらも影響を受けないが，フラグは次のように設定される．

CMP　CL,DH

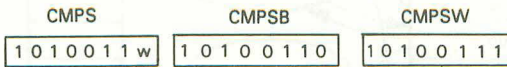
サイクル数：　レジスタとレジスタ：3

　　　　　　　メモリとレジスタ　：9＋EA

　　　　　　　レジスタとメモリ　：16＋EA

CMPS/CMPSB/CMPSW　(Compare String)

メモリとメモリを比較する.

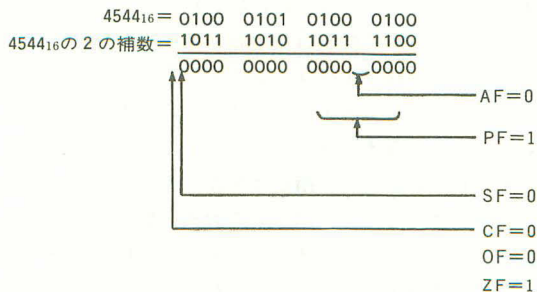
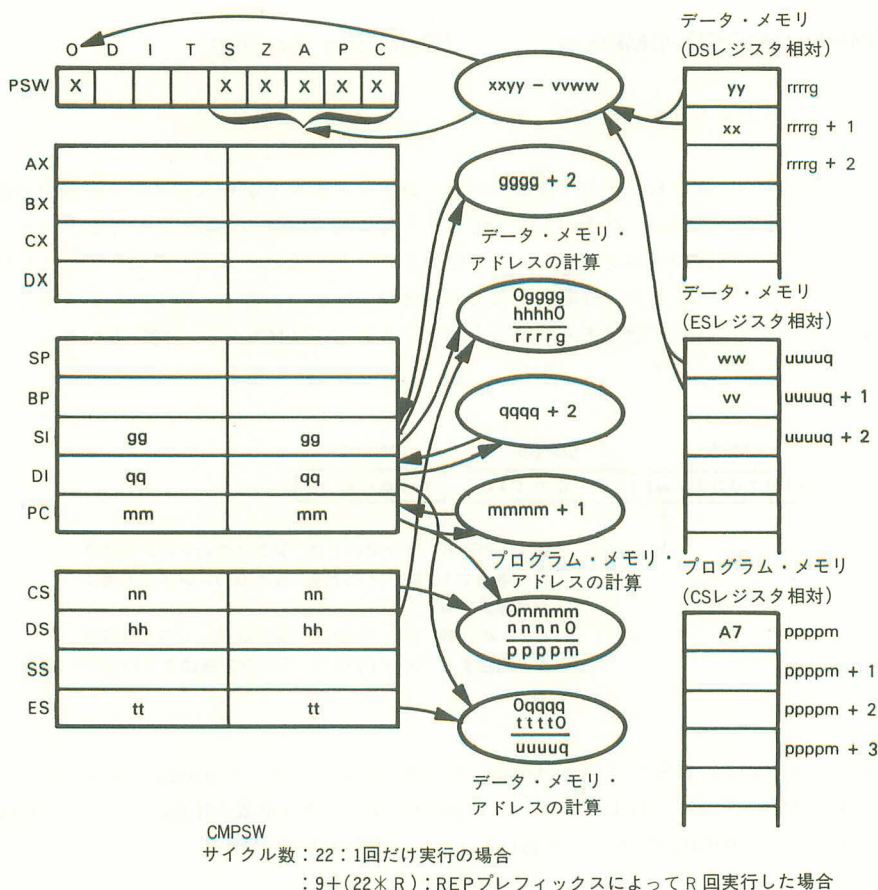
ＳＩレジスタで示されるメモリ位置の内容とＤＩレジスタで示されるメモリ位置の内容の比較を行なう．比較は，ＳＩレジスタで示されるメモリ位置の内容からＤＩレジスタで示されるメモリ位置の内容を減算し，その結果をフラグに設定することによって行なわれる．減算に用いられるメモリ位置のどちらも影響を受けない．ＳＩとＤＩのレジスタは，ＤＦフラグの値に応じて増加あるいは減少する．８あるいは16ビットの指定ができる．

命令コードを次に示す．

ＤＦフラグが0で，ＤＳレジスタが0600_{16}を含み，ＳＩレジスタが0108_{16}を含み，ＥＳレジスタが0060_{16}を含み，ＤＩレジスタが0188_{16}を含み，メモリ位置06108_{16}のワードが4544_{16}で，メモリ位置00788_{16}のワードが4544_{16}であると仮定する．

CMPSW

の実行後，ＳＩレジスタは$010A_{16}$に，ＤＩレジスタは$018A_{16}$となり，フラグは次のように設定される．

CMPSW
サイクル数：22：1回だけ実行の場合
：9＋(22＊R)：REPプレフィックスによってR回実行した場合

注)

1．REPプレフィックスやLOCKプレフィックスはこの命令と共に用いられる．この命令と共に，LOCKプレフィックスとREPプレフィックスの両方が用いられた場合は問題となる．この件についての解説は第4章を参照のこと．

2．SIレジスタによって示されるオペランドのデフォルト・セグメント・レジスタはDSである．このセグメント・レジスタは，セグメント変更プレフィックスを用いて変えられる．DIによって示されるオペランドのデフォルト・セグメント・レジスタはESである．このセグメント・レジスタの指定は無効にならない．

3．インテルのアセンブラは，バイトとワードの形式に加えて，汎用の形式を許している．汎用の形式に対して，アセンブラは，8ビットあるいは16ビットのどちらの比較が行なわれるのかを決められる情報を必要とする．これがどのように行なわれるかについての説明は，この章の最後を参照のこと．

4．ＲＥＰプレフィックスを伴ったＣＭＰＳの実行時間は，次のように表わせる．

$$\underbrace{\text{REP}}_{2+}\quad\underbrace{\text{CMPS}}_{9+(22*R)}$$

もしＲ＝10ならば，実行時間は231クロック・サイクルとなる．

CWD　(Convert Word to Doubleword)

ＡＸレジスタの符号をＤＸレジスタに拡張する．

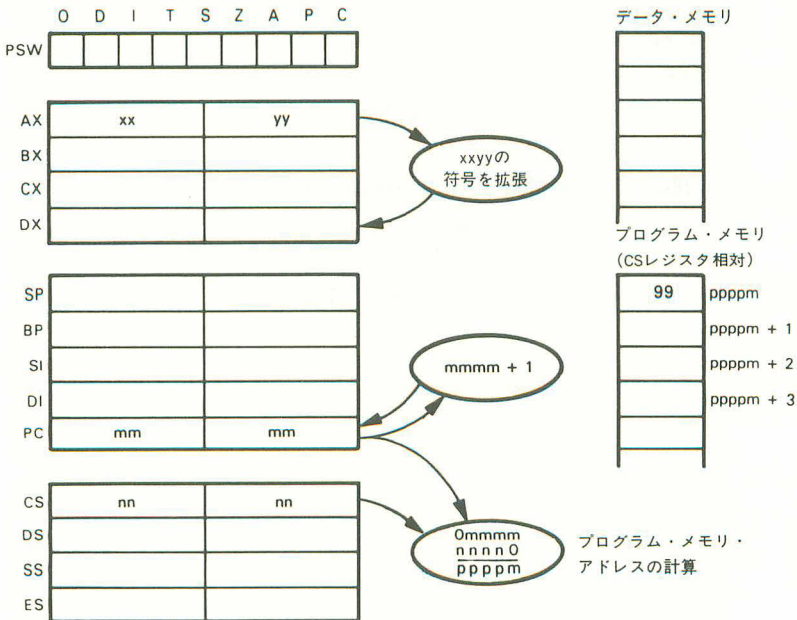
ＡＸレジスタの最上位ビットが１ならば，$FFFF_{16}$をＤＸレジスタにストアし，そうでなければ0000_{16}をＤＸレジスタにストアする．

命令コードを次に示す．

$$\underbrace{\text{CWD}}_{99}$$

ＡＸレジスタが$B001_{16}$を含むと仮定する．

CWD

の実行後，ＤＸレジスタは$FFFF_{16}$を含む．

CWD
サイクル数：5

注)

1．ステータスは影響を受けない．

2．この命令は除算を行なうときに有用である．16ビットの除数が用いられるならば，32ビットの被除数が必要となる．ＡＸレジスタにのみ有効なビットが存在するならば，この命令は32ビットの被除数を得るために符号ビットをＤＸレジスタに拡張する．この方法はＩＤＩＶ命令に最適であることに注意．

DAA (Decimal Adjust for Addition)

加算後のアキュムレータの内容を10進数に変換する．

ＡＬレジスタの内容を，２進化10進（ＢＣＤ）の形式に変換する．この命令は，２つのＢＣＤの加算後にだけ用いられる．すなわち，ＡＤＤ　ＤＡＡあるいはＡＤＣ　ＤＡＡを，ＢＣＤの結果を得るためにＢＣＤのソース・オペランドを操作する複合の10進演算命令と見なす．

変換のアルゴリズムを次に示す．

1．ＡＦフラグが１，あるいはＡＬレジスタの下位４ビットがＡからＦまでの値ならば，ＡＬレジスタに06_{16}を加算し，ＡＦフラグを１にする．

2．ＣＦフラグが１，あるいはＡＬレジスタの上位４ビットが９よりも大きければ，ＡＬレジスタに60_{16}を加算し，ＣＦフラグを１にする．

命令コードを次に示す．

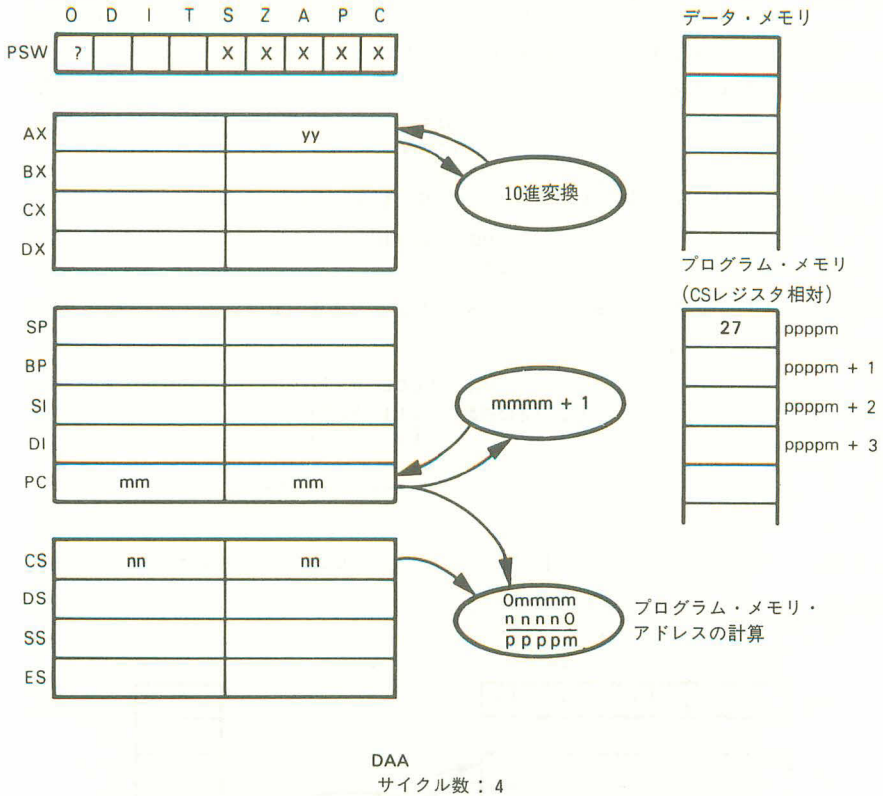
```
DAA
 27
```

ＡＬレジスタが28_{16}を含み，ＢＬレジスタが68_{16}を含むと仮定する．

```
ADD   AL, BL
DAA
```

の実行後，ＡＬレジスタは，90_{16}ではなくて96_{16}となる．

DAA
サイクル数：4

注）

1．この命令は，2つのパック形式BCDオペランドの加算に有用である．2つのパック形式オペランドの減算の変換については，DAS命令を参照．ASCIIの加算と減算の結果の変換については，AAAとAASの命令を参照．

DAS　(Decimal Adjust for Subtraction)

減算後のアキュムレータの内容を10進数に変換する．

この命令は，ALレジスタの内容を2進化10進（BCD）の形式に変換する．また，この命令は，2つのBCDの減算後にだけ用いられる．すなわち，SUB　DASあるいはSBB　DASを，BCDの結果を得るためにBCDのソース・オペランドを操作する複合の10進演算命令と見なす．

変換のアルゴリズムを次に示す．

1．AFフラグが1，あるいはALレジスタの下位4ビットがAからFの間ならば，ALレジスタから06_{16}を減算し，AFフラグを1にする．

2．ＣＦフラグが1，あるいはＡＬレジスタの上位4ビットが9より大きければ，ＡＬレジスタから60_{16}を減算し，ＣＦフラグを1にする．

命令コードを次に示す．

DAS
2F

ＡＬレジスタが86_{16}を含み，ＡＨレジスタが07_{16}を含むと仮定する．

```
SUB   AL, AH
DAS
```

の実行後，ＡＬレジスタは79_{16}となる．ＳＵＢ命令の結果，ＡＬレジスタは$7F_{16}$を含んでいる．

86_{16} ＝1000 0110
07_{16}の2の補数＝1111 1001
0111 1111
CF＝0

ＡＬレジスタの下位4ビットはＦに等しいので，アルゴリズムの第1のステップが実行される．ＡＦフラグは1に設定される．

DAS
サイクル数：4

注)

1．これは2つのパック形式ＢＣＤに対する10進減算のアジャストメント・アルゴリズムである．減算のアジャストメントに有効な他の操作にＡＡＳ命令があり，これはＡＳＣＩＩの減算結果のアジャストを行なう．

DEC　mem/reg　　　(Decrement)

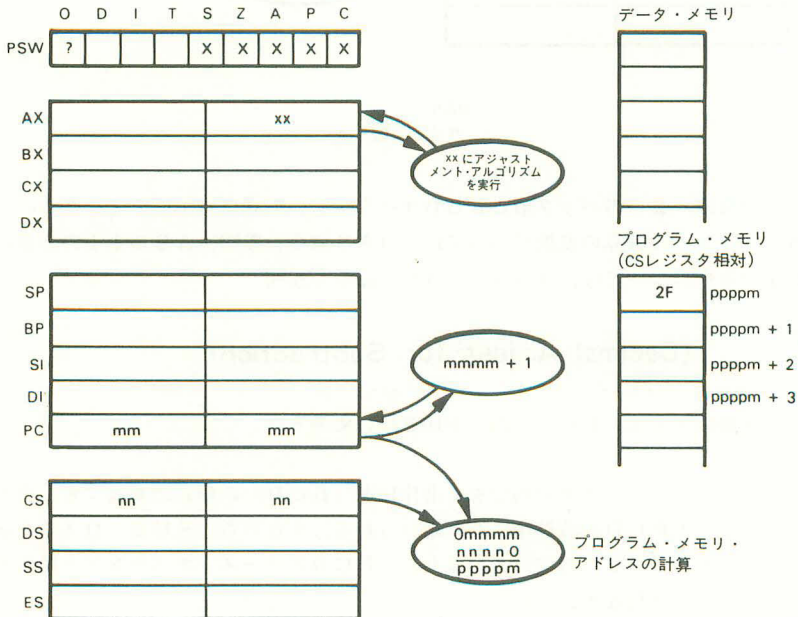
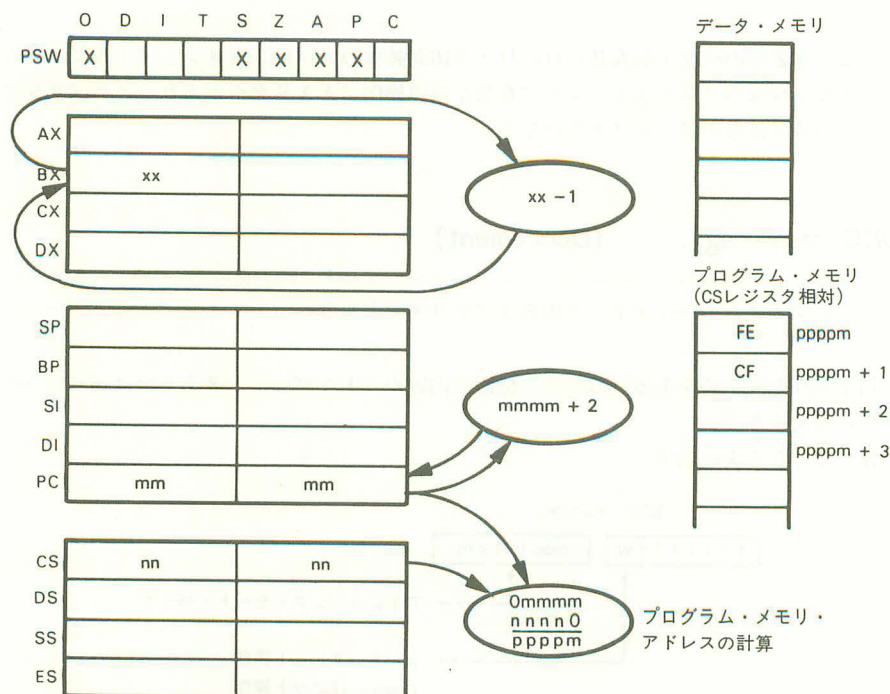
レジスタあるいはメモリの内容をデクリメントする．

指定されたレジスタあるいはメモリ位置の内容から1を減じる．8あるいは16ビットの操作が指定できる．

命令コードを次に示す．

ＢＨレジスタが$4F_{16}$を含むと仮定する．

DEC　BH

の実行後，ＢＨレジスタは$4E_{16}$となる．

DEC BH
サイクル数：レジスタ・オペランド：3
メモリ・オペランド：15+EA

注)

1. この命令は，8080の命令 DCR reg と同じ機能を果たす．種々のアドレッシング・モードが利用可能で8あるいは16ビットが選択できることから，この命令は8080の命令と比較して非常に有力であることに注意．
2. この命令でセグメント・レジスタはデクリメントできない．
3. この命令は，通常16ビット・レジスタをデクリメントするためには用いられない．命令 DEC reg はこの機能を果たし，しかもプログラム・メモリが1バイトですむ．この命令は，8ビット・レジスタあるいはメモリをデクリメントするために用いられる．
4. この命令は，キャリー・フラグに影響を与えない．

DEC reg (Decrement)

レジスタをデクリメントする．

指定されたレジスタの内容から1を減じる．これは16ビットのデクリメント命令である．命令コードを次に示す．

例として，CXレジスタが0200_{16}を含む場合を考える．

DEC　CX

の実行結果，CXレジスタの内容は$01FF_{16}$に減少している．

DEC　CX
サイクル数：2

注）

1．この命令は，8080の命令　DCX reg と同じ機能を果たす．

2．セグメント・レジスタは，この命令を用いてデクリメントされない．

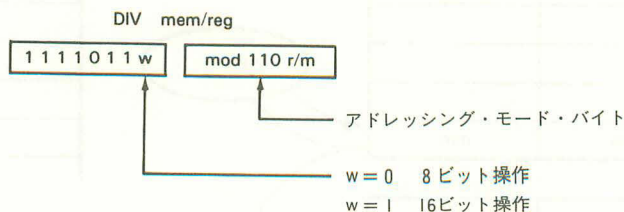
DIV mem/reg　　(Divide)

ＡＨ：ＡＬあるいはＤＸ：ＡＸを，レジスタあるいはメモリの内容で割る．

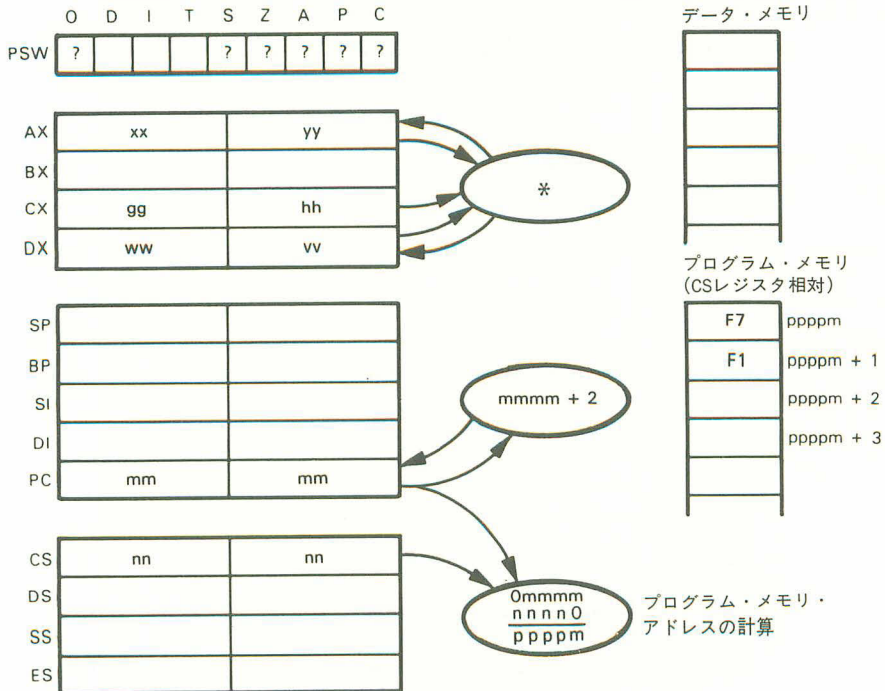
両方のオペランドを符号なし２進級と見なして，ＡＨ：ＡＬ（８ビット操作）あるいはＤＸ：ＡＸ（16ビット操作）のレジスタを，指定されたレジスタあるいはメモリ位置の内容で割る．16ビットの符号なし数値を８ビットの符号なし数値で割ること，あるいは32ビットの符号なし数値を16ビットの符号なし数値で割ることが可能である．８ビット操作の場合は，８ビットの商はＡＬレジスタに返され，８ビットの余りはＡＨレジスタに返される．もしＡＬレジスタに返される商が FF_{16} より大きいと，タイプ０（０による除算）のインタラプトが発生する．16ビットの操作では，16ビットの商はＡＸレジスタに返され，16ビットの余りはＤＸレジスタに返される．もしＡＸレジスタに返される商が $FFFF_{16}$ より大きいと，タイプ０（０による除算）のインタラプトが発生する．

０による除算のインタラプトの結果，次の動作が行なわれる．

1．フラグ・レジスタをスタックにプッシュする．
2．ＩＦとＴＦのフラグをクリヤする．
3．ＣＳレジスタをスタックにプッシュする．
4．メモリ位置 00002_{16} のワードをＣＳレジスタにロードする．
5．ＰＣをスタックにプッシュする．
6．メモリ位置 00000_{16} のワードをＰＣレジスタにロードする．

命令コードを次に示す．

例として，ＡＸレジスタが $0F05_{16}$ を含み，ＤＸレジスタが $068A_{16}$ を含み，ＣＸレジスタが $08E9_{16}$ を含む場合を考える．

DIV　CX

の実行後，ＡＸレジスタは商 $BBE1_{16}$，DXレジスタは余り $073C_{16}$ となる．ＯＦ，ＳＦ，ＺＦ，ＡＦ，ＰＦ，ＣＦのフラグの値は，この操作で不定となる．すなわち，ＤＩＶ命令後の特定のフラグがとる値は分からない．

＊wwvvxxyyがgghhによって割られる．
商はＡＸに戻され，余りはＤＸに戻される．

DIV CX

サイクル数：16ビットのメモリによる除算：(150－168)＋EA
8ビットのメモリによる除算：(86－96)＋EA
16ビットのレジスタによる除算：144－162
8ビットのレジスタによる除算：80－90

注)

1．この命令の実行後，算術演算のフラグの値は不定となる．

2．ＤＩＶ命令の実行以前に，ＤＩＶ命令が結果として0による除算のインタラプトを発生するかを知る必要があるならば，次に示す一連の命令は有用である．

16ビット除算：ＣＬが除数を含むと仮定．

```
CMP AH, CL
JNB OVERFLOW$HANDLER
```

32ビット除算：ＢＸが除数を含むと仮定．

```
CMP DX, BX
JNB OVERFLOW$HANDLER
```

この種類のチェックは，0による除算のインタラプト処理が十分でない場合に有用と

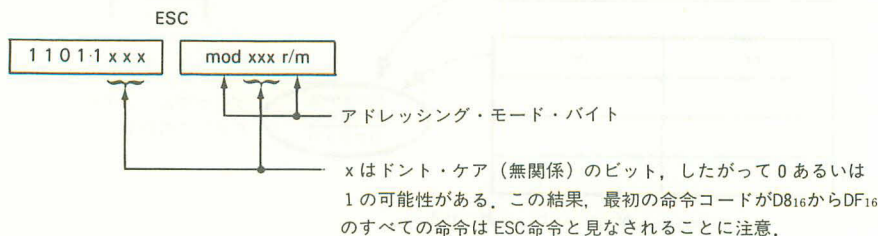
なる．

3．被除数と除数が同一の長さのときは，最初に被除数はＣＢＷあるいはＣＷＤの命令を用いて16ビットあるいは32ビットに拡張しておかなければならない．

ESC mem (Escape)

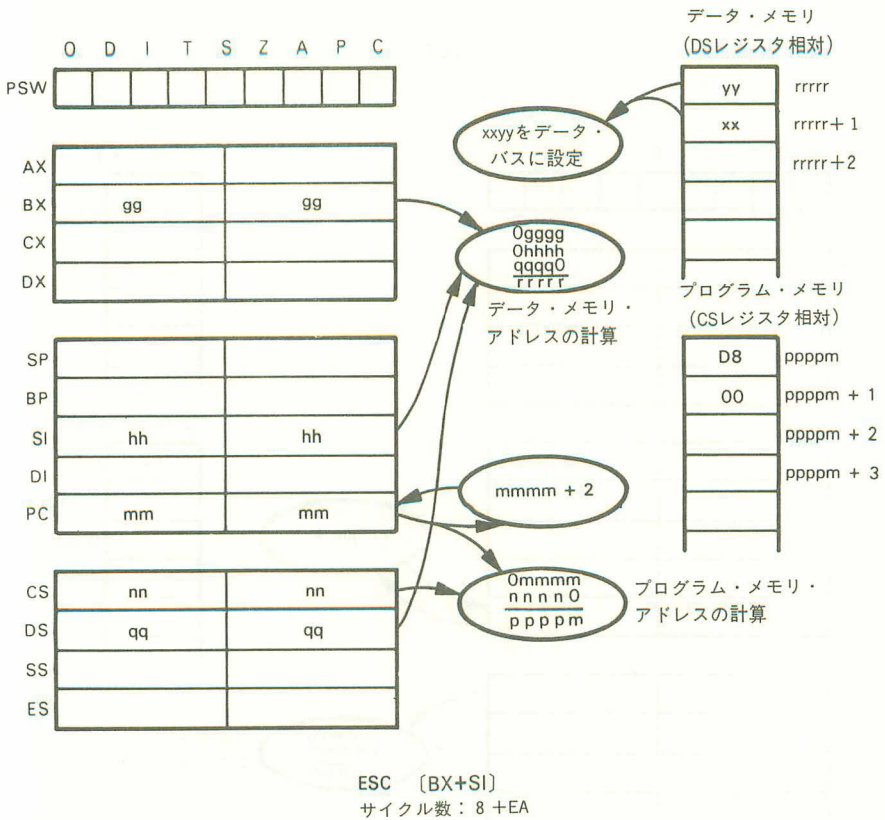
メモリにアクセスする．

この命令は，指定されたメモリ位置の内容を，データ・バスに設定する．本質的には，8086に関する限りこの命令は何の操作も行なわない．この命令は，他のプロセッサに8086のアドレッシング・モードを利用し，8086の命令の流れから命令を受け取らせるために用いられる．

命令コードを次に示す．

ＢＸレジスタが 063A$_{16}$を含み，ＳＩレジスタが0003$_{16}$を含み，ＤＳレジスタが FF80$_{16}$ を含み，メモリ位置 FFE3D$_{16}$ のワードがＣ308$_{16}$であるとする．

ESC 〔BX+SI〕

を実行すると，指定されたメモリ素子によってＲＥＡＤＹラインが確立された時点で，データ・ライン上にはＣ308$_{16}$のデータが存在する．

ESC 〔BX+SI〕
サイクル数：8 +EA

注)

1．mod=11(すなわち，レジスタを指定)の場合は，この命令は無操作となり，CLOCK CYCLES＝2である．

HLT　　(Halt)

プロセッサをホルト状態にする．

ＨＬＴ命令が実行されると，プログラムの実行は停止する．実行を再開するには，外部インタラプトあるいはリセットが必要となる．レジスタあるいはステータスは影響を受けない．

注意：ＨＬＴ命令以前にＳＴＩ命令によってインタラプトが許可されていなければ，8086 ＣＰＵは，ハードウエアのリセットあるいはノンマスカブル・インタラプトによる場合を除いて，ホルト状態から抜け出すことができない．

命令コードを次に示す.

HLT
サイクル数: 2

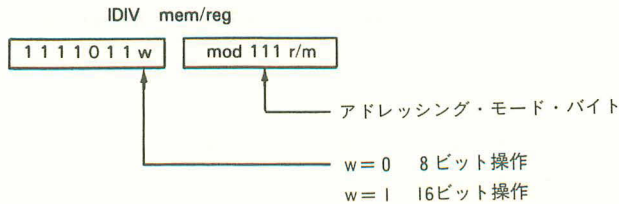
IDIV　mem/reg　　　(Integer Divide)

ＡＨ：ＡＬあるいはＤＸ：ＡＸを，レジスタあるいはメモリの内容で割る．

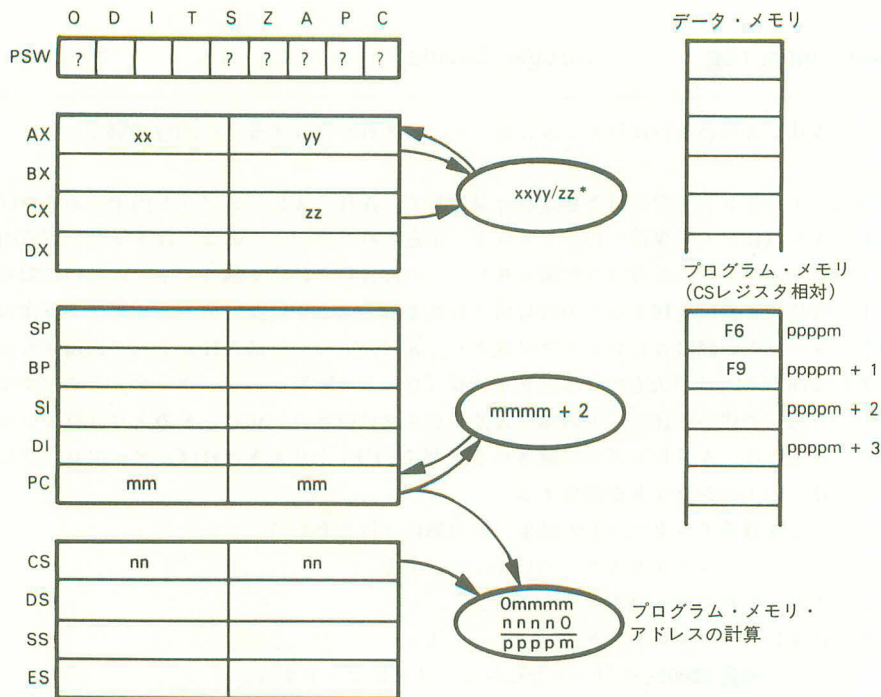
両方のオペランドを符号付き2進数と見なして，ＡＨ：ＡＬ（8ビット操作）あるいはＤＸ：ＡＸ（16ビット操作）のレジスタを，指定されたレジスタあるいはメモリ位置の内容で割る．16ビットの符号付き数値を8ビットの符号付き数値で割ること，あるいは32ビットの符号付き数値を16ビットの符号付き数値で割ることが可能である．8ビット操作の場合，8ビットの商はＡＬレジスタに返され，8ビットの余りはＡＨレジスタに返される．返される商が$7F_{16}$より大きければ，タイプ0（0による除算）のインタラプトが発生する．16ビット操作の場合，16ビットの商はＡＸレジスタに返され，16ビットの余りはＤＸレジスタに返される．ＡＸレジスタに返される商が$7FFF_{16}$より大きければ，タイプ0（0による除算）のインタラプトが発生する．

0による除算のインタラプトの結果，次の動作が行なわれる．

1．フラグ・レジスタをスタックにプッシュする．
2．ＩＦとＴＦのフラグをクリヤする．
3．ＣＳレジスタをスタックにプッシュする．
4．メモリ位置00002_{16}のワードをＣＳレジスタにロードする．
5．ＰＣをスタックにプッシュする．
6．メモリ位置00000_{16}のワードをＰＣレジスタにロードする．

命令コードを次に示す．

ＣＬレジスタが$0D_{16}$を含み，ＡＸレジスタが$00A9_{16}$を含むと仮定する．

IDIV　　CL

の実行後，ＡＸレジスタは$000D_{16}$となる．

＊商はＡＬレジスタに戻し，余りはＡＨレジスタに戻す.

IDIV　CL

サイクル数：８ビットのメモリによる除算：(107—118)+EA
16ビットのメモリによる除算：(171—190)+EA
８ビットのレジスタによる除算：101—112
16ビットのレジスタによる除算：165—184

注)

1．これは，符号付き数値の除算命令である．両方のオペランドは，次の範囲の符号付き２進数として取り扱われる．

８ビット操作：＋127〜－128

16ビット操作：＋32767〜－32768

符号なしの除算については，ＤＩＶ命令を参照．

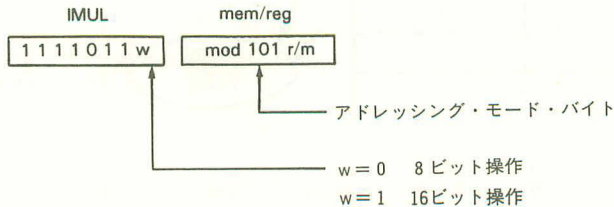
2．この命令の実行後，フラグの値は不定となる．

3．被除数と除数が同一の長さのときは，最初に被除数は，ＣＢＷあるいはＣＷＤの命令を用いて，16あるいは32ビットの形に変換されなければならない．

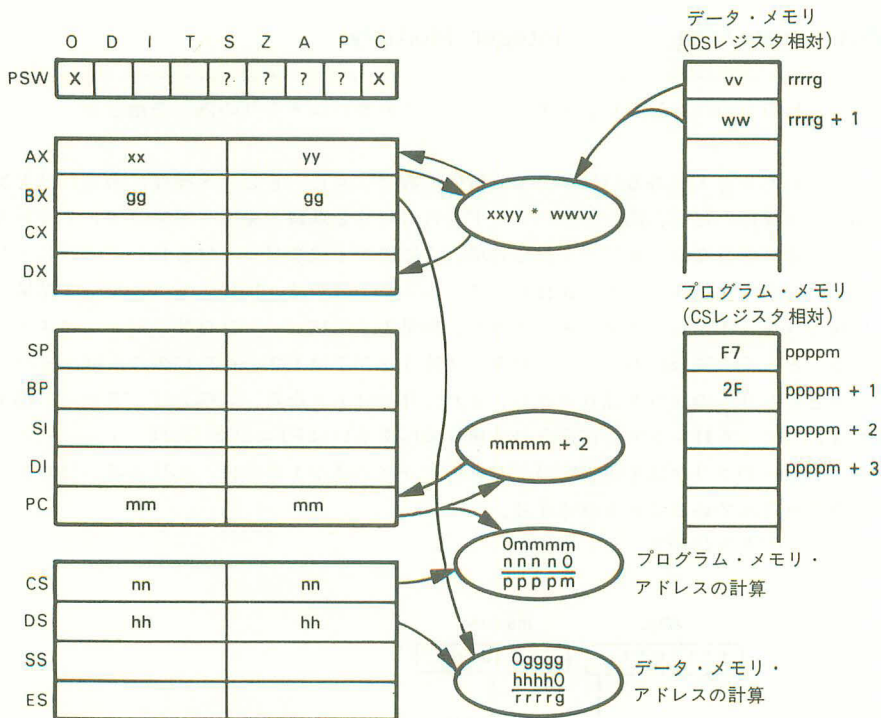
IMUL　mem/reg　　　(Integer Multiply)

ALあるいはAXのレジスタに，レジスタあるいはメモリの内容を乗じる．

指定されたレジスタあるいはメモリ位置の内容と，AL（8ビット操作）あるいはAX（16ビット操作）とを，両方のオペランドを符号付き2進数と見なして乗じる．すなわち，符号付き乗算を行なう．8ビット操作の場合，結果の下位8ビットはALレジスタにストアされ，結果の上位8ビットはAHレジスタにストアされる．16ビット操作では，結果の下位16ビットはAXレジスタにストアされ，結果の上位16ビットはDXレジスタにストアされる．どちらの場合においても，結果の上位1/2が下位1/2の符号拡張ならばオーバーフローとキャリーのフラグは0となり，それ以外では1となる．（例えば，8ビット操作が行なわれて，AHレジスタに返される値が00_{16}あるいはFF_{16}でなければ，キャリーとオーバーフローのフラグは1になる.）これらのフラグの値が1ならば，AHあるいはDXは有効な桁を含んでいることを意味する．

命令コードを次に示す．

例として，AXレジスタが$04E8_{16}$を含み，DSレジスタが0100_{16}を含み，BXレジスタが0006_{16}を含み，メモリ位置01006_{16}のワードが$4E20_{16}$の場合を考える．

IMUL　AX,〔BX〕

の実行後，AXレジスタは$4D00_{16}$に，DXレジスタは$017F_{16}$となり，キャリーとオーバーフローのフラグは1になる．

IMUL AX,[BX]
サイクル数：8ビットのメモリによる乗算：(86−104)＋EA
16ビットのメモリによる乗算：(34−160)＋EA
8ビットのレジスタによる乗算：80—98
16ビットのレジスタによる乗算：128—154

注)
1．これは符号付き数値の乗算操作である．両方のオペランドは次の範囲の数値として取り扱われる．

8ビット操作：＋127〜−128

16ビット操作：＋32767〜−32768

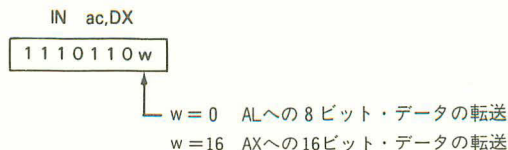
符号なしの乗算操作については，命令ＭＵＬを参照．

2．ある場合には，乗算を行なうためにシフトを用いることがより適当となるときがある．このような場合は，メモリの保存が最も重要ではなく速さが必要なときに生じる．

3．この命令の実行後，ＳＦ，ＺＦ，ＡＦ，ＰＦのフラグは不定となる．

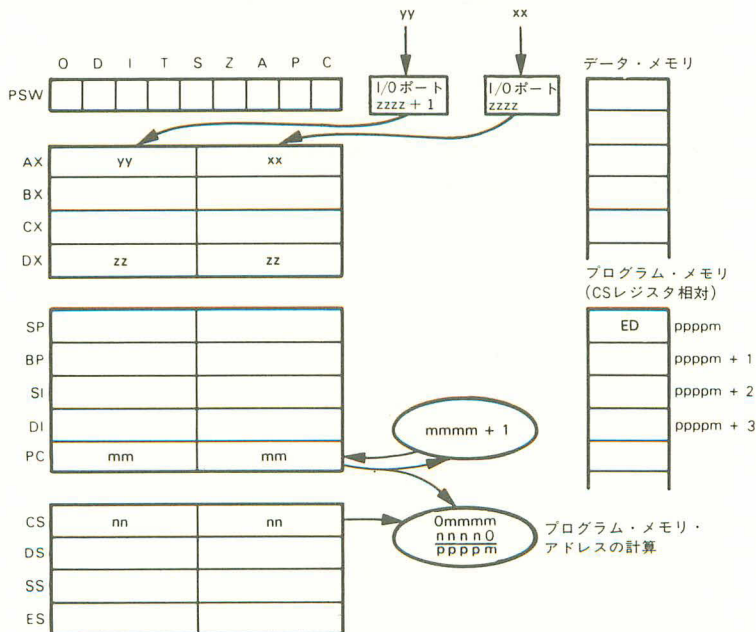
IN ac, DX (Input)

アキュームレータへの入力を行なう.

この命令は，DXレジスタの内容で指定されるI/Oポートから，AL(8ビット転送)あるいはAX(16ビット転送)のレジスタへ，8あるいは16ビットのデータ要素をロードする．命令コードを次に示す．

他のレジスタ（ALあるいはAXを除く）あるいはステータスは，影響を受けない．

DXレジスタが1234_{16}を含み，ポート1234_{16}のI/Oバッファが23_{16}を含み，ポート1235_{16}のI/Oバッファが$F4_{16}$を含むと仮定する．

IN AX,DX

を実行すると，ALレジスタに23_{16}が，AHレジスタに$F4_{16}$がロードされる．

IN AX,DX
サイクル数：8

注)

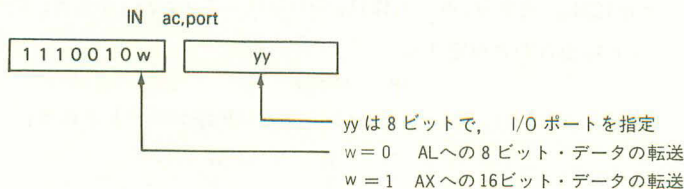
1．この命令で，0からFFFF$_{16}$までのアドレスを割り当てられている入力ポートにアクセスすることができる．

2．acとして，8ビットに対してはALが，16ビットに対してはAXのみが指定できる．

IN　ac,port　　　　(Input)

アキュームレータへの入力を行なう．

この命令は，命令の2番目のバイトで指定されるI/OポートからAL（8ビット転送）あるいはAX（16ビット転送）のレジスタへ8あるいは16ビットのデータ要素をロードする．

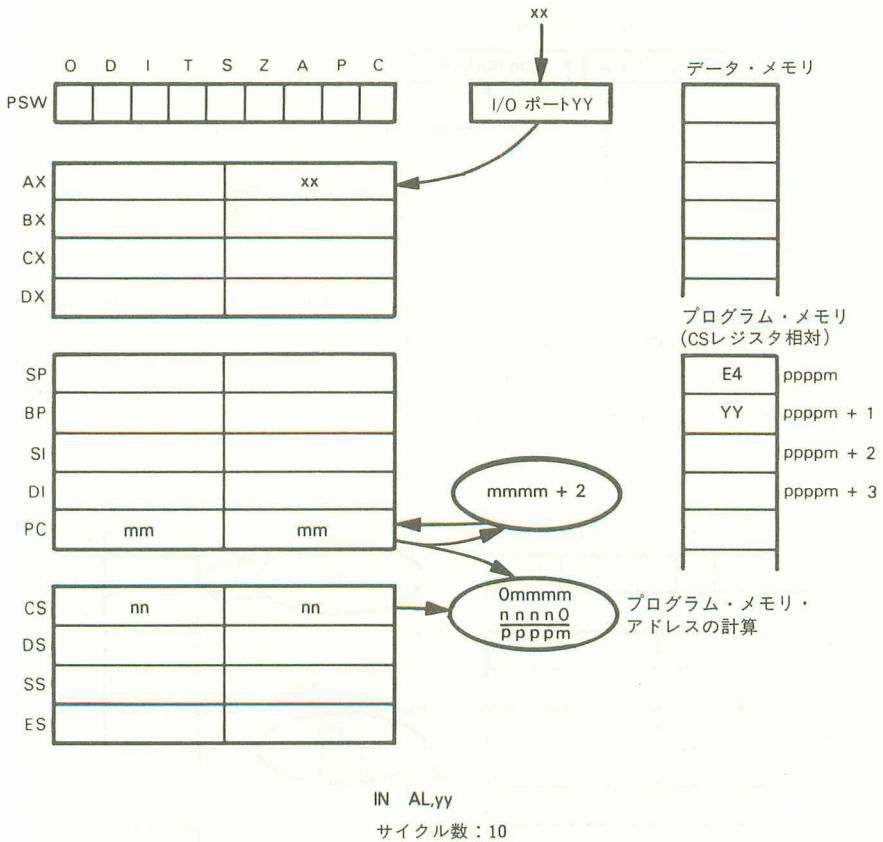
命令コードを次に示す．

他のレジスタ（ALあるいはAXを除く）あるいはステータスは影響を受けない．

ポート06$_{16}$のI/Oバッファが43$_{16}$を含むと仮定する．

IN　　AL,06H

の実行によって，ALレジスタに43$_{16}$がロードされる．

IN　AL,yy

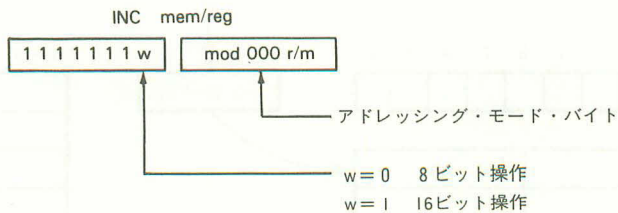
サイクル数：10

注)

1．この命令で，0からFF$_{16}$までのアドレスを割り当てられているI/Oポートにアクセスすることができる．この範囲外のアドレスのポート指定は，命令　IN　ac,DXを参照．
2．この命令は，8080の命令　IN　port　と同じ機能を果たす．
3．acとして，8ビットに対してはALが，16ビットに対してはAXのみが指定できる．

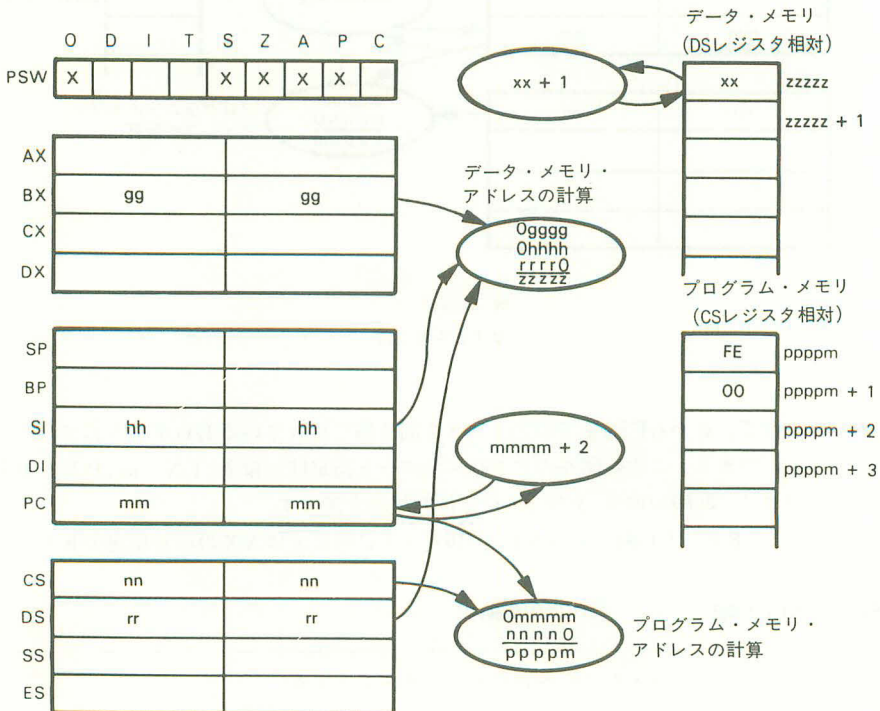
INC　mem/reg　　　(Increment)

レジスタあるいはメモリの内容をインクリメントする．

指定されたレジスタあるいはメモリ位置の内容に1を加える．8あるいは16ビットの操作が指定できる．

命令コードを次に示す．

ＤＳレジスタが$F800_{16}$を含み，ＢＸレジスタの内容が0280_{16}で，ＳＩレジスタが$001E_{16}$を含み，メモリ位置$F829E_{16}$が64_{16}の場合を考える．

INC　〔BX＋SI〕

の実行後，メモリ位置$F829E_{16}$は65_{16}を含む．

INC　[BX + SI]
サイクル数：メモリ・オペランド：15＋EA
　　　　　　レジスタ・オペランド：3

注)

1．セグメント・レジスタはこの命令でインクリメントはできない．

2．この命令は，8080の命令　ＩＮＲ reg と同じ機能を果たす．またこの命令は，8080の命令と比較して非常に有力であることに注意．

3．この命令は，通常16ビットのレジスタをインクリメントするためには用いられない．命令　ＩＮＣ reg はこの機能を果たし，しかもプログラム・メモリが1バイトですむ．この命令は，8ビット・レジスタあるいはメモリをインクリメントするために用いられる．

4．この命令は，キャリー・フラグに影響を与えない．

INC　reg　　　　(Increment)

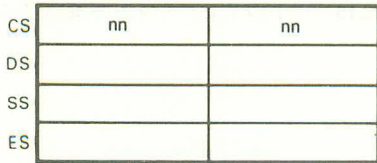
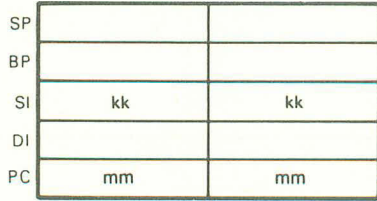
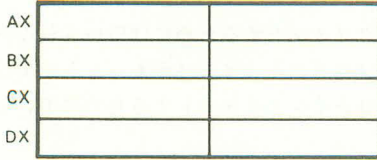
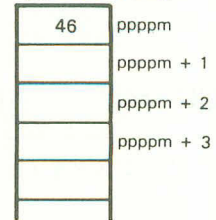
レジスタをインクリメントする．

指定されたレジスタの内容に1を加える．これは16ビットのインクリメント命令である．

rrr＝000：ＡＸ
001：ＣＸ
010：ＤＸ
011：ＢＸ
100：ＳＰ
101：ＢＰ
110：ＳＩ
111：ＤＩ

ＳＩレジスタの内容が00FF$_{16}$の場合を考える．

INC　SI

の実行によって，ＳＩレジスタの内容は0100$_{16}$に増加する．

INC SI
サイクル数:2

注)

1．この命令は，8080の命令　INX reg と同じ機能を果たす．

2．セグメント・レジスタは，この命令を用いてインクリメントすることはできない．

3．この命令は，キャリー・フラグに影響を与えない．

INT　　(Interrupt)

ソフトウエアでインタラプトを発生させる.

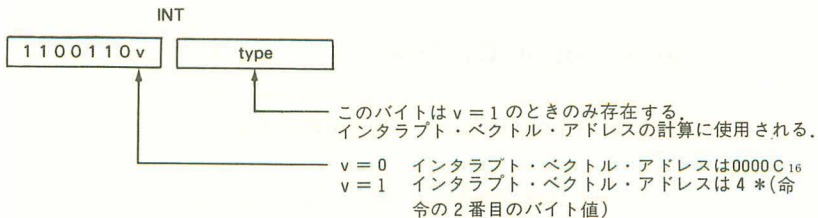
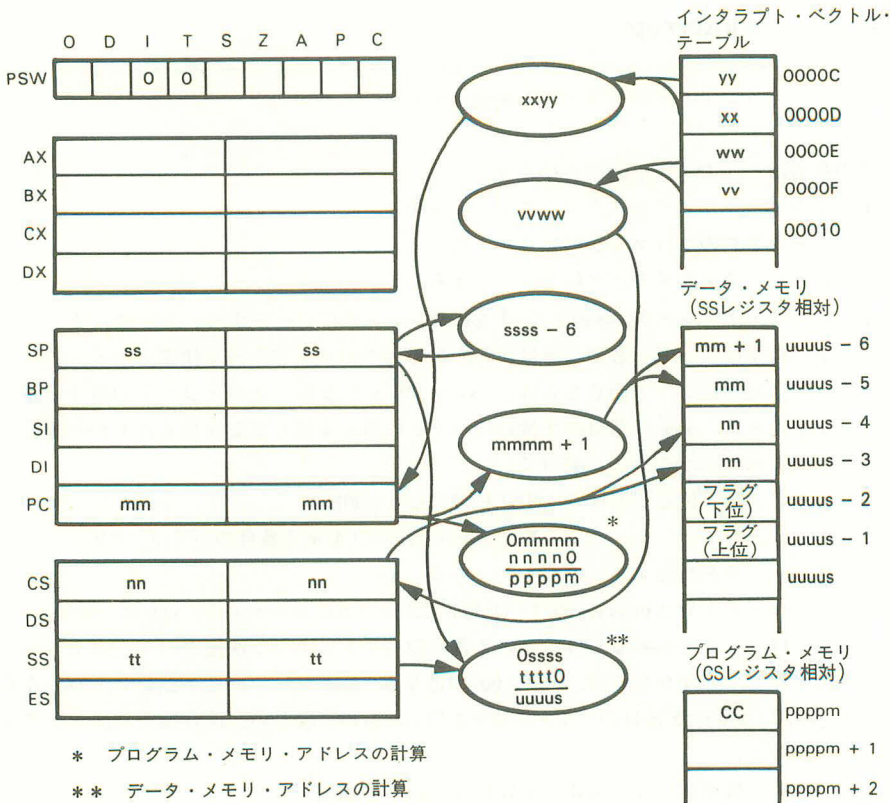
この命令は, 次の一連の動作を行なう.

1. フラグ・レジスタをスタックにプッシュする.
2. ＩＦとＴＦのフラグを0にクリヤする.
3. ＣＳレジスタをスタックにプッシュする.
4. メモリ・アドレス$00xxx_{16}$のワードをＣＳレジスタにロードする. xxxは, オペ・コードの最下位ビットと必要ならば命令の2番目のバイトによって決定される. オペ・コードの最下位ビットが0ならば, xxxは$00E_{16}$となる. オペ・コードの最下位ビットが1ならば, xxxは命令の2番目のバイトの値を4倍して2を加えたものに等しい. 言い換えれば, 次のように書ける.

   ＩＦ　最下位ビット＝0　ＴＨＥＮ　xxx ＝$00E_{16}$
   　　　　　　　　　　　ＥＬＳＥ　xxx ＝（4 ＊ 2番目のバイト）＋2
5. ＰＣレジスタをスタックにプッシュする.
6. メモリ・アドレス$00yyy_{16}$のワードをＰＣレジスタにロードする. yyyは, 命令コードの最下位ビットと必要ならば命令の2番目のバイトによって決定される. 命令コードの最下位ビットが0ならば, yyyは$00C_{16}$となる. 命令コードの最下位ビットが1ならば, yyyは命令の2番目のバイトの値を4倍したものに等しい. 言い換えれば, 次のように書ける.

   ＩＦ　最下位ビット＝0　ＴＨＥＮ　yyy ＝$00C_{16}$
   　　　　　　　　　　　ＥＬＳＥ　yyy ＝4 ＊ 2番目のバイト

命令コードを次に示す.

INT
サイクル数：52：v＝0のとき
　　　　　　51：v＝1のとき

INTO　　(Interrupt if Oveflow)

ＯＦ＝１ならば，タイプ4のインタラプトを発生させる．

ＯＦ＝０ならば，この命令は無操作となる．ＯＦ＝１ならば，次の一連の動作を行なう．

1．フラグ・レジスタをスタックにプッシュする．

2．ＩＦとＴＦのフラグを０に設定する．

3．ＣＳレジスタをスタックにプッシュする．

4．メモリ位置00012_{16}のワードをＣＳレジスタに移動する．

5．ＰＣレジスタをスタックにプッシュする．

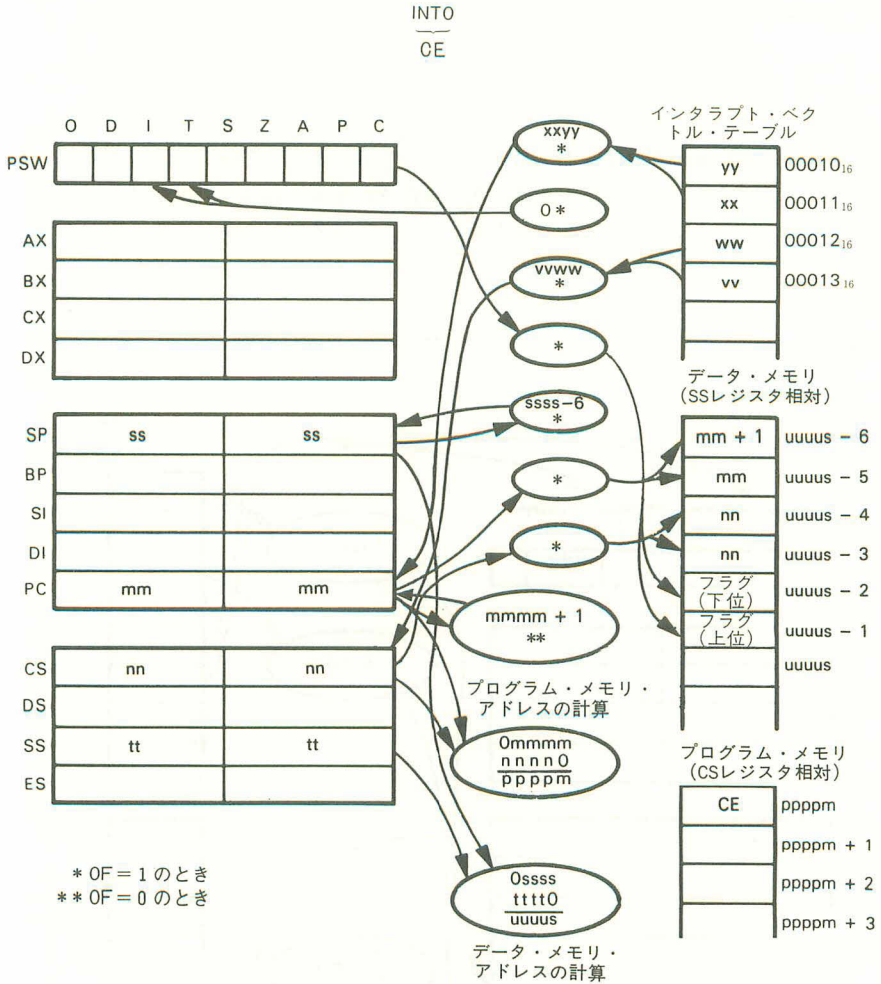
6．メモリ位置00010_{16}のワードをＰＣレジスタに移動する．

この位置から実行を再開する．

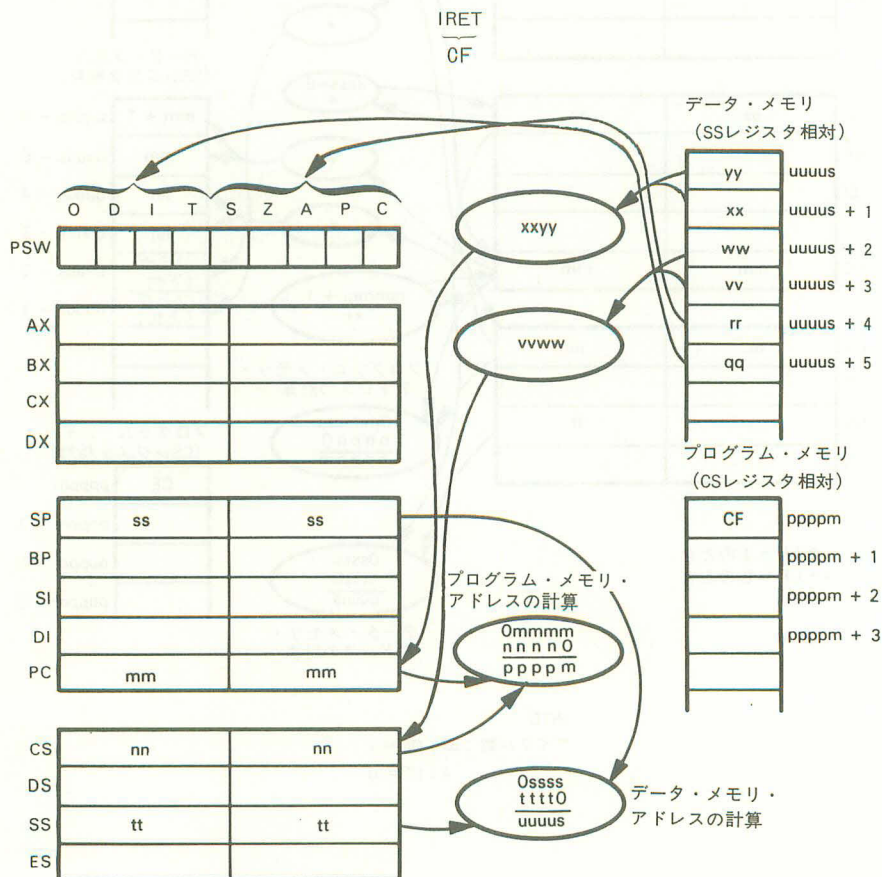
命令コードを次に示す．

INTO

サイクル数：53：OF＝1

4：OF＝0

IRET　　(Interrupt Return)

インタラプト処理からリターンする．

スタックのトップの2バイトをプログラム・カウンタにポップする．この2バイトは，次に実行される命令のオフセット・アドレスを与える．スタックの次の2バイトをCSレジスタにポップする．この2バイトは，次に実行される命令のコード・セグメント・アドレスを与える．スタックの次の2バイトをフラグ・レジスタにポップする．以前のプログラム・カウンタ，コード・セグメントとフラグのレジスタの内容は失なわれる．

命令コードを次に示す．

IRET
CF

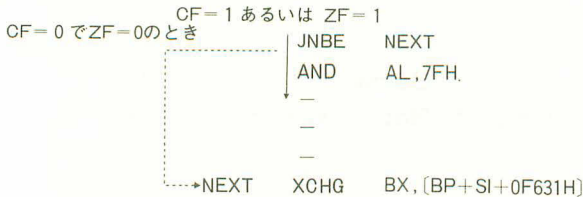
IRET
サイクル数：32

JA disp (Jump if Above)
JNBE disp (Jump if Not Below nor Equal)

上ならばジャンプする．／下でも等しくもなければジャンプする．

この命令は，キャリー・フラグとゼロ・フラグが共に0の場合だけジャンプが行なわれ，それ以外では次の命令が実行されることを除いて，JMP disp 命令と同じである．

命令コードを次に示す．

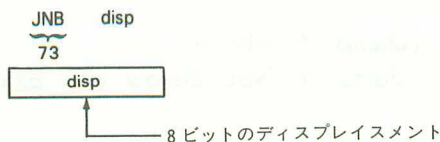
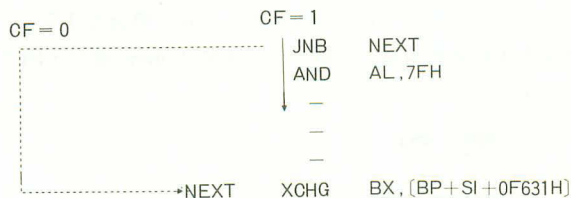
次の一連の命令において，

JNBE命令に続いて，キャリー・フラグとゼロ・フラグが0ならばXCHG命令が実行される．キャリー・フラグあるいはゼロ・フラグが1ならば，AND命令が実行される．

サイクル数：分岐したとき：16
　　　　　　分岐しなかったとき：4

JAE disp (Jump if Above or Equal)
JNB disp (Jump if Not Below)

上あるいは等しければジャンプする．／下でなければジャンプする．

この命令は，キャリー・フラグが0の場合だけジャンプが行なわれ，それ以外では次の命令が実行されることを除いて，JMP disp 命令と同じである．

命令コードを次に示す．

次の一連の命令において，

ＪＮＢ命令に続いて，キャリー・フラグが０ならばＸＣＨＧ命令が実行される．キャリー・フラグが１ならば，ＡＮＤ命令が実行される．

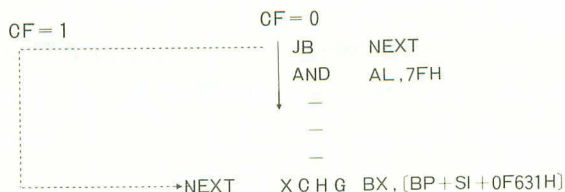
サイクル数：分岐したとき：16
分岐しなかったとき：4

JB disp (Jump if Below)
JNAE disp (Jump if Not Above nor Equal)

下ならばジャンプする．／上でも等しくもなければジャンプする．

この命令は，キャリー・フラグが１の場合だけジャンプが実行されることを除けば，ＪＭＰ命令と同じである．

命令コードを次に示す．

次の一連の命令において，

ＪＢ命令に続いて，キャリー・フラグが１ならばＸＣＨＧ命令が実行される．キャリー・フラグが０の場合はＡＮＤ命令が実行される．

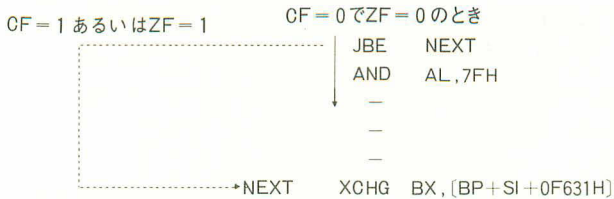
サイクル数：分岐したとき：16
　　　　　　分岐しなかったとき：４

JBE　disp　　(Jump if Below or Equal)
JNA　disp　　(Jump if Not Above)

下あるいは等しければジャンプする．／上でなければジャンプする．

この命令は，キャリー・フラグあるいはゼロ・フラグが１ならばジャンプし，そうでなければ次の命令を実行することを除いて，ＪＭＰ disp 命令と同じである．

命令コードを次に示す．

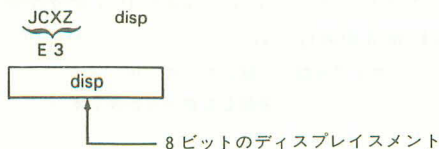
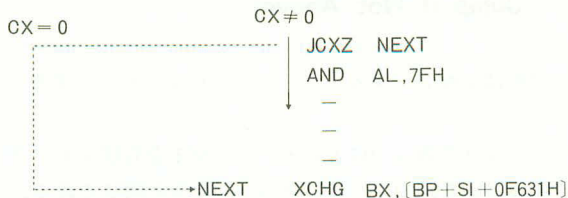
次の一連の命令において，

ＪＢＥ命令に続いて，キャリー・フラグあるいはゼロ・フラグが１ならばＸＣＨＧ命令が実行される．キャリー・フラグとゼロ・フラグが共に０ならば，ＡＮＤ命令が実行される．

サイクル数：分岐したとき：16
　　　　　　分岐しなかったとき：４

JCXZ　disp　　(Jump if CX is Zero)

ＣＸが０ならばジャンプする．

この命令は，ＣＸレジスタが０の場合だけジャンプし，それ以外では次の命令を実行することを除いて，ＪＭＰ disp 命令と同じである．

命令コードを次に示す．

次の一連の命令において，

ＪＣＸＺ命令に続いて，ＣＸレジスタが０ならばＸＣＨＧ命令が実行される．ＣＸレジスタが０でなければ，ＡＮＤ命令が実行される．

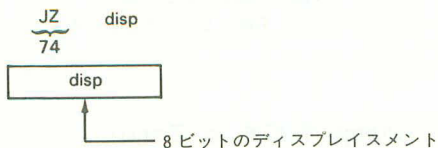
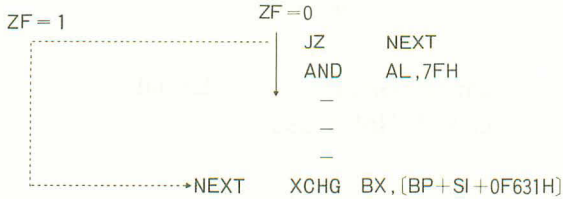
この命令はＣＸの０を判断するためにゼロ・フラグを参照するのではなく，ＣＸレジスタを直接参照することに注意．

サイクル数：分岐した時：18
　　　　　　分岐しなかったとき：6

JE disp (Jump if Equal)
JZ disp (Jump if Zero)

等しければジャンプする．／０ならばジャンプする．

この命令は，ゼロ・フラグが１ならばジャンプし，それ以外では次の命令を実行することを除いて，ＪＭＰ disp 命令と同じである．

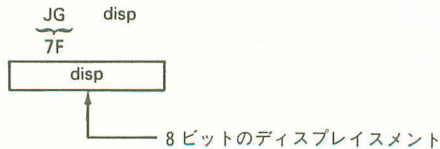
命令コードを次に示す．

次の一連の命令において，

ＪＺ命令に続いて，ゼロ・フラグが1ならばＸＣＨＧ命令が実行される．ゼロ・フラグが0ならば，ＡＮＤ命令が実行される．

サイクル数：分岐したとき：16
　　　　　　分岐しなかったとき：4

JG　　disp　　(Jump if Greater)
JNLE　disp　　(Jump if Not Less nor Equal)

大きければジャンプする．／小さくもなく等しくもなければジャンプする．

この命令は，ゼロ・フラグが0でしかもサイン・フラグとオーバーフロー・フラグが等しければジャンプし，それ以外では次の命令を実行することを除いて，ＪＭＰ disp 命令と同じである．

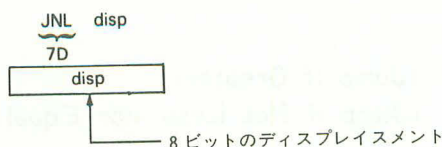
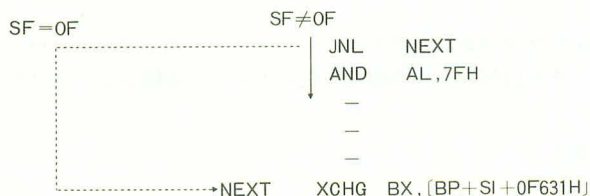
命令コードを次に示す．

次の一連の命令において，

ＪＧ命令に続いて，ゼロ・フラグが0でサイン・フラグとオーバーフロー・フラグが等しければ，ＸＣＨＧ命令が実行される．ゼロ・フラグが1あるいはサイン・フラグとオーバーフロー・フラグが等しくなければ，ＡＮＤ命令が実行される．

サイクル数：分岐したとき：16
分岐しなかったとき：4

JGE disp (Jump if Greater or Equal)
JNL disp (Jump if Not Less)

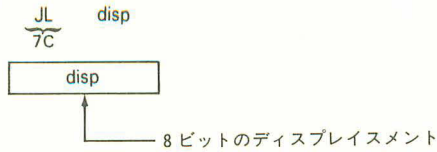
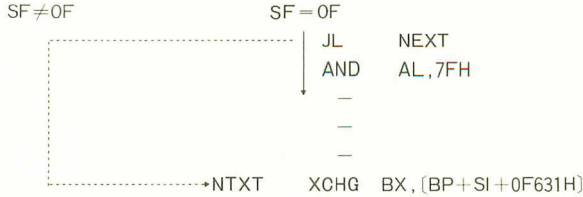
大きいか等しければジャンプする．／小さくなければジャンプする．

この命令は，サイン・フラグがオーバーフロー・フラグに等しければジャンプし，それ以外では次の命令を実行することを除いて，JMP disp 命令と同じである．
命令コードを次に示す．

次の一連の命令において，

JNL命令に続いて，サイン・フラグがオーバーフロー・フラグに等しければXCHG命令が実行される．サイン・フラグがオーバーフロー・フラグに等しくなければ，AND命令が実行される．

サイクル数：分岐したとき：16
分岐しなかったとき：4

JL disp (Jump if Less)
JNGE disp (Jump if Not Greater nor Equal)

小さければジャンプする．／大きくもなく等しくもなければジャンプする．

この命令は，サイン・フラグがオーバーフロー・フラグに等しくなければジャンプし，それ以外では次の命令を実行することを除いて，JMP disp 命令と同じである．
命令コードを次に示す．

次の一連の命令において，

ＪＬ命令実行後，サイン・フラグがオーバーフロー・フラグに等しくなければＸＣＨＧ命令が実行される．サイン・フラグとオーバーフロー・フラグが等しければ，ＡＮＤ命令が実行される．

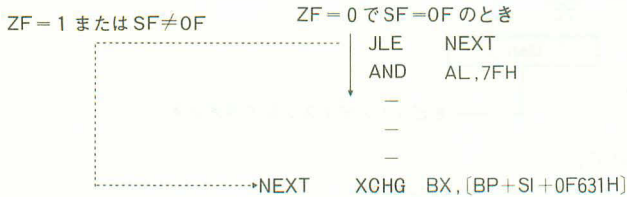
サイクル数：分岐したとき：16
　　　　　　分岐しなかったとき：4

JLE　disp　(Jump if Less or Equal)
JNG　disp　(Jump if Not Greater)

小さいか等しければジャンプする．／大きくなければジャンプする．

この命令は，ゼロ・フラグが1かあるいはサイン・フラグがオーバーフロー・フラグに等しくなければジャンプし，それ以外では次の命令を実行することを除いて，ＪＭＰ disp命令と同じである．

命令コードを次に示す．

次の一連の命令において，

ＪＬＥ命令に続いて，ゼロ・フラグが１あるいはサイン・フラグがオーバーフロー・フラグに等しくなければ，ＸＣＨＧ命令が実行される．ゼロ・フラグが０でサイン・フラグがオーバーフロー・フラグに等しければ，ＡＮＤ命令が実行される．

サイクル数：分岐したとき：16
　　　　　　分岐しなかったとき：4

JMP addr (Jump)

オペランドで指定された位置にジャンプする．

後続のプログラム・メモリの２バイトの内容をＰＣレジスタへ移動する．その次のプログラム・メモリの２バイト（命令のバイト４と５）の内容をＣＳレジスタに移動する．この位置から実行を続ける．以前のプログラム・カウンタとコード・セグメント・レジスタの内容は失なわれる．

命令コードを次に示す．

JMP addr
サイクル数：15

注）

1．イミディエイトの32ビットは目標のアドレスで，遠い位置のラベル（セグメント・レジスタを変える）と考えられる．

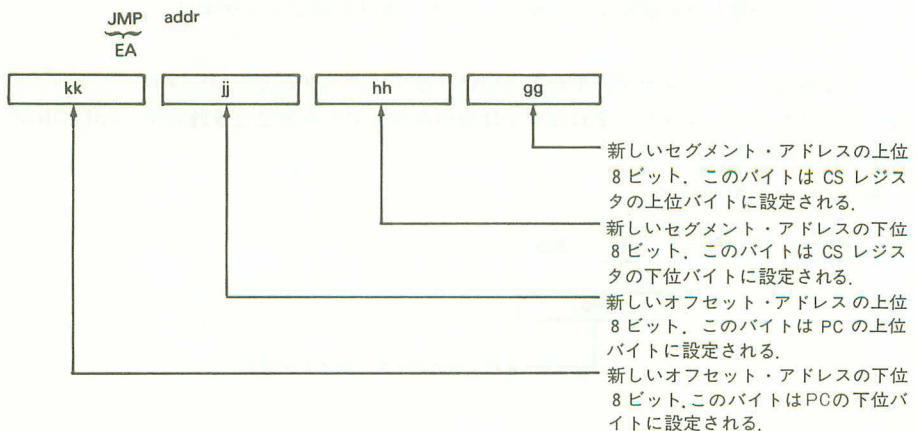
2．遠い位置のラベルへ直接の移動が行なわれる．

JMP　disp　　　(Jump)

オペランドで指定された位置にジャンプする．

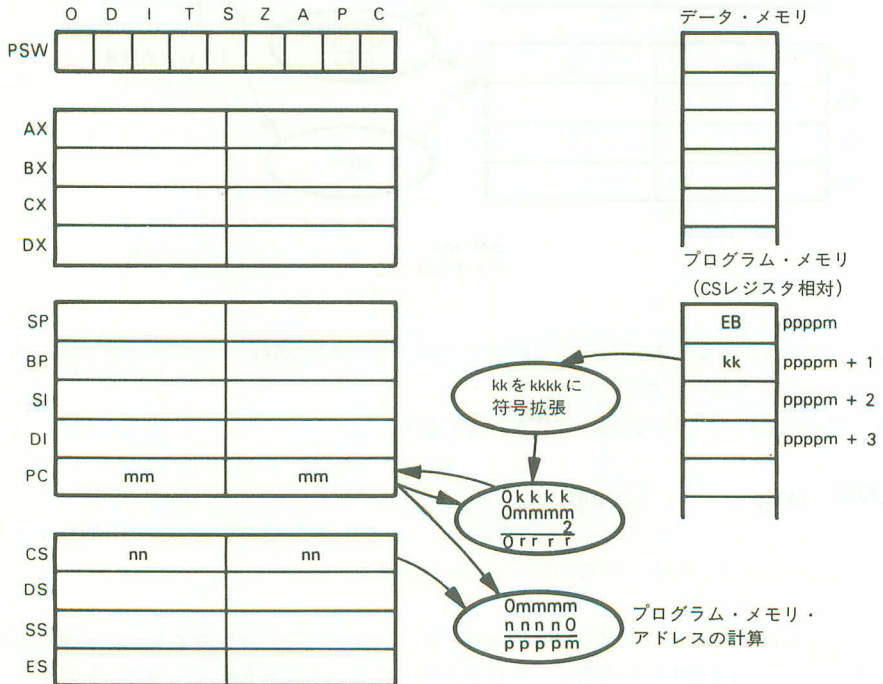
この命令は，2番目のオブジェクト・コード・バイト（符号付きの8ビット・ディスプレイスメントと見なす）の内容を，プログラム・カウンタに2を加えた内容に加算する*．これは，次に実行される命令のオフセット・アドレスになる．以前のプログラム・カウンタの内容は失なわれる．コード・セグメント・レジスタの内容は変化しない．

命令コードを次に示す．

＊　2を加えるのは，この命令が2バイトで構成されているため（訳者注）．

次の一連の命令において，

```
        JMP    NEXT
        AND    AL，7FH
        —
        —
        —
NEXT    XOR    AL，7FH
```

JMP命令に続いて，XOR命令が実行される．AND命令は，どこかでジャンプあるいはコールの命令でこの命令に分岐しない限りは決して実行されない．

O D I T S Z A P C
PSW
AX
BX
CX
DX
SP
BP
SI
DI
PC mm mm
CS nn nn
DS
SS
ES
データ・メモリ
プログラム・メモリ
(CSレジスタ相対)
EB ppppm
kk ppppm + 1
ppppm + 2
ppppm + 3
kkをkkkkに
符号拡張
0kkkk
0mmmm
2
0rrrr
0mmmm
nnnn0
ppppm
プログラム・メモリ・
アドレスの計算

JMP kk
サイクル数：15

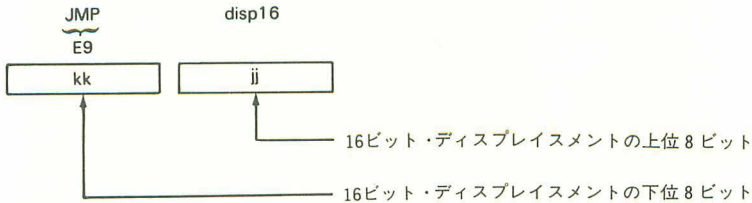
注)

1．この命令は，プログラム相対アドレス指定を用いている．これは，An Introduction to Microcomputers：Volume I —Basic Concepts（Osborne/McGraw-Hill, 1978）に述べられているプログラム相対ページングと類似している．異なっているのは，8ビットの符号付きディスプレイスメントが加算される前に，プログラム・カウンタの内容は次の命令を指すために増加することである．
2．これは，現在のセグメント内の自己相対の分岐である．
3．8ビットのディスプレイスメントは近い位置のラベルのディスプレイスメントと考えられる．

JMP　disp16　　　(Jump)

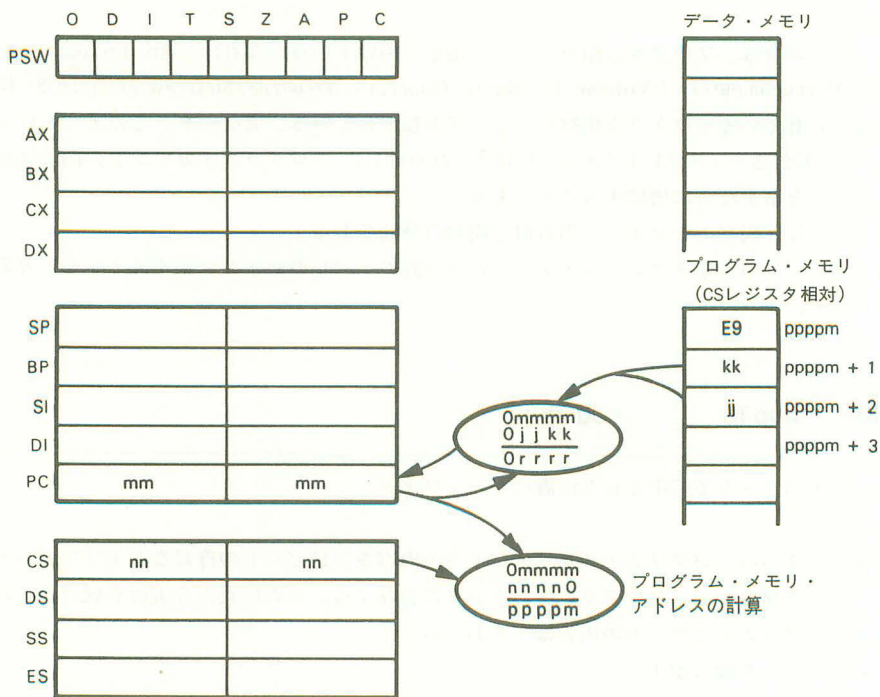
オペランドで指定された位置にジャンプする．

後続の2つのプログラム・メモリ・バイトの内容を，16ビットの符号なしディスプレイスメントと見なして，プログラム・カウンタに加算する．この位置から実行を続ける．以前のプログラム・カウンタの内容は失なわれる．

命令コードを次に示す．

次の一連の命令において，

```
        JMP    NEXT
BRICKS  AND    AL,7FH
        —
        —
        —
NEXT    STOS   BYTE
```

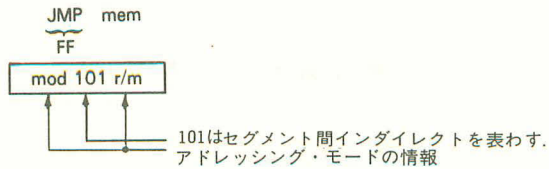
ＪＭＰ命令実行後，ＳＴＯＳ命令が実行される．ＡＮＤ命令は，処理のどこかでＣＡＬＬあるいはＪＭＰ命令で BRICKS がそのオペランドとして参照されない限りは，実行されることはない．

JMP　jjkk
サイクル数：15

注)
1．これは，現在のセグメント内での自己相対の分岐である．
2．16ビットのディスプレイスメントは，近い位置のラベル（このセグメント内）のディスプレイスメントと考えられる．

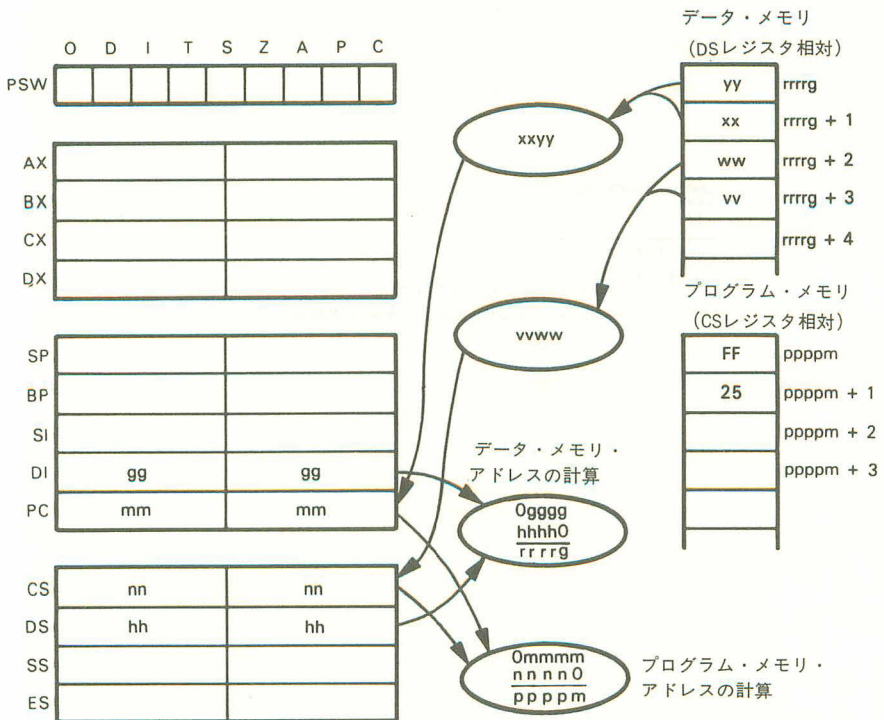
JMP　mem　　　(Jump)

オペランドで指定された位置にジャンプする．

指定されたメモリ位置のワードをプログラム・カウンタへ移動し，続くワードをCSレジスタに移動する．この位置から実行を続ける．以前のプログラム・カウンタとコード・セグメント・レジスタの内容は失なわれる．

命令コードを次に示す．

ＤＳレジスタが7000_{16}を含み，ＤＩレジスタが0404_{16}を含み，メモリ位置70404_{16}のワードが1000_{16}でメモリ位置70406_{16}のワードが$7E00_{16}$であると仮定する．

JMP far-ptr〔DI〕

の実行後，プログラム・カウンタは1000_{16}に，ＣＳレジスタは$7E00_{16}$になる．命令の実行は$7F000_{16}$から続けられる．

JMP far_ptr[DI]
サイクル数：24+EA セグメント間

注）

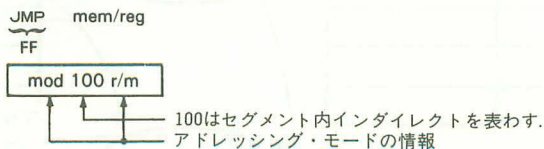
1．この命令では，レジスタ・アドレッシングは有効でない．

2．これは，メモリを用いたセグメント間（セグメントを変える）インダイレクト・ジャンプであり，ジャンプ・テーブルによく用いられる．
3．32ビットの目的アドレスは，遠い位置のラベルと考えられる．
4．〔ＤＩ〕の前に far-ptr ディレクティブを書くことによって，アセンブラはメモリ中の32ビット・ポインタを用いるＪＭＰ命令を生成する．

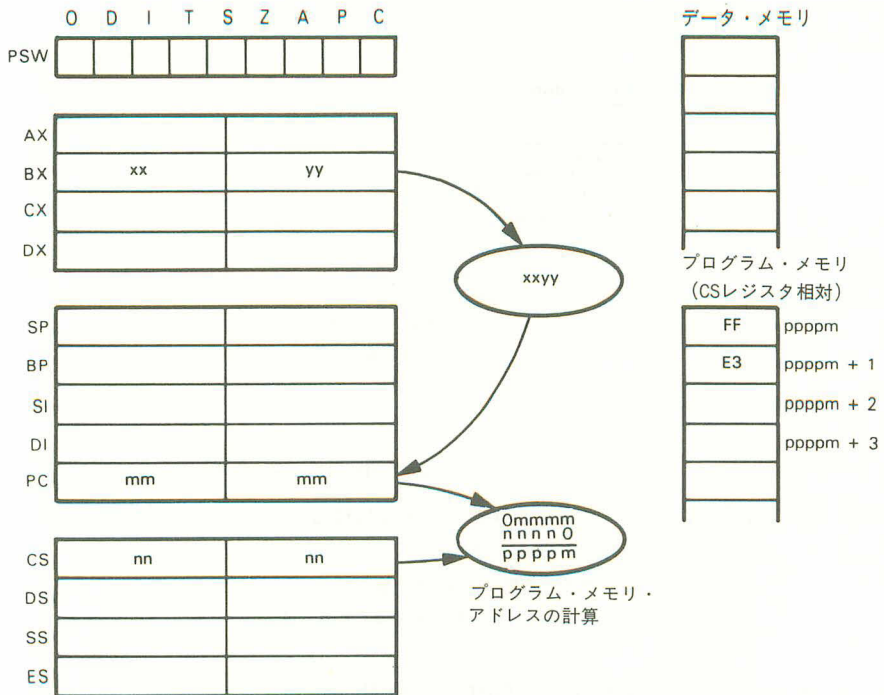
JMP mem/reg (Jump)

オペランドで指定された位置にジャンプする．

指定されたオペランドがレジスタならば，そのレジスタの内容をプログラム・カウンタに移動する．指定されたオペランドがメモリ位置ならば，そのメモリ位置の内容をプログラム・カウンタに移動する．この位置から実行を続ける．以前のプログラム・カウンタの内容は失なわれる．ＣＳレジスタは変化しない．

命令コードを次に示す．

ＢＸレジスタが $14A9_{16}$ を含むと仮定する．

JMP BX

の実行後，ＰＣは $14A9_{16}$ となり，$14A9_{16}$ を次の命令のオフセット・アドレスとして，実行を再開する．

JMP　BX

サイクル数：JMP BX:　11：レジスタによる場合

JMP [BX]: 16 + EA：メモリによる場合

注)

1．これは，メモリあるいはレジスタによるセグメント内インダイレクト・ジャンプである．

2．レジスタあるいはステータスは影響を受けない．

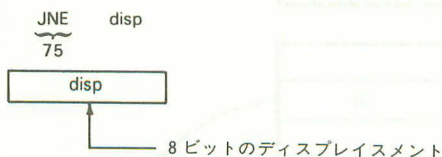
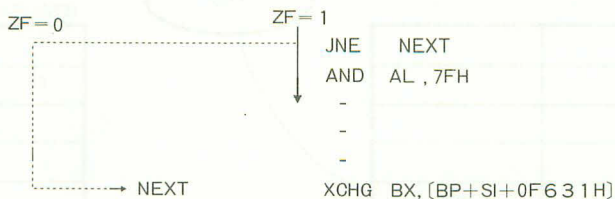
3．16ビットの目的アドレスは，近い位置のラベル（セグメント内）と考えられる．

4．far_ptr がないことで，前のＪＭＰ命令のときの32ビット・ポインタではなくて，メモリの16ビット・ポインタを表わしている．このアセンブラの変換で，1つのニーモニックＪＭＰをさまざまな2進命令コードに用いることができる．

JNE　disp　　(Jump if Not Equal)
JNZ　disp　　(Jump if Not Zero)

等しくなければジャンプする．／0でなければジャンプする．

この命令は，ゼロ・フラグが0ならばジャンプし，それ以外では次の命令を実行するこ

とを除いて，ＪＭＰ disp 命令と同じである．

命令コードを次に示す．

次の一連の命令において，

ＪＮＥ命令に続いて，ゼロ・フラグが０ならばＸＣＨＧ命令が実行される．ゼロ・フラグが１ならば，ＡＮＤ命令が実行される．

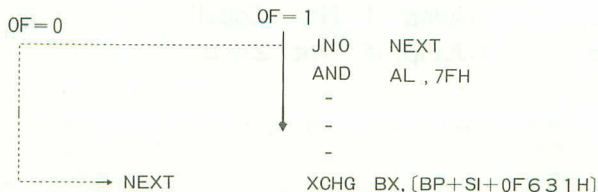
サイクル数：分岐したとき：16
　　　　　　分岐しなかったとき：4

JNO disp (Jump if Not Overflow)

オーバーフローでなければジャンプする．

この命令は，オーバーフロー・フラグが０ならばジャンプし，それ以外では次の命令を実行することを除いて，ＪＭＰ disp 命令と同じである．

命令コードを次に示す．

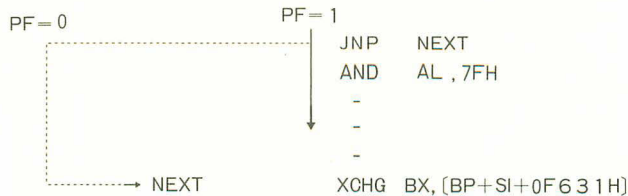
次の一連の命令において，

JNO命令に続いて，オーバーフロー・ステータスが0ならば，XCHG命令が実行される．オーバーフロー・ステータスが1ならば，AND命令が実行される．

サイクル数：分岐したとき：16
分岐しなかったとき：4

JNP disp　(Jump if Not Parity)
JPO disp　(Jump if Parity Odd)

パリティ・フラグが0ならばジャンプする．／パリティが奇数ならばジャンプする．

この命令は，パリティ・フラグが0ならばジャンプし，それ以外では次の命令を実行することを除いて，JMP disp 命令と同じである．

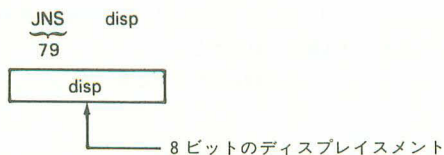
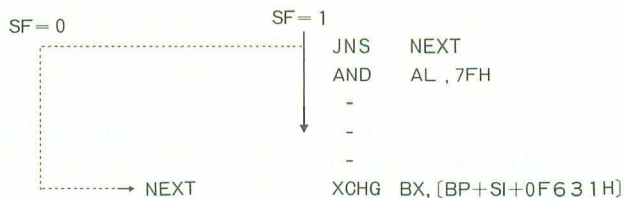
命令コードを次に示す．

次の一連の命令において，

JNP命令に続いて，パリティ・フラグが0ならばXCHG命令が実行される．パリティ・フラグが1ならば，AND命令が実行される．

サイクル数：分岐したとき：16
分岐しなかったとき：4

JNS disp　(Jump if Not Sign)

サイン・フラグが0ならばジャンプする．

この命令は，サイン・フラグが0ならばジャンプし，それ以外では次の命令を実行することを除いて，JMP disp 命令と同じである．

命令コードを次に示す.

次の一連の命令において,

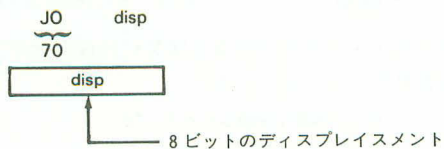
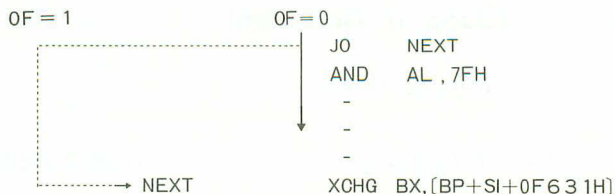
ＪＮＳ命令実行後，サイン・フラグが0ならばＸＣＨＧ命令が実行され，そうでなければＡＮＤ命令が実行される.

サイクル数：分岐したとき：16
分岐しなかったとき：4

JO disp (Jump if Overflow)

オーバーフローならばジャンプする.

この命令は，オーバーフロー・フラグが1ならばジャンプし，それ以外では次の命令を実行することを除いて，ＪＭＰ disp 命令と同じである.

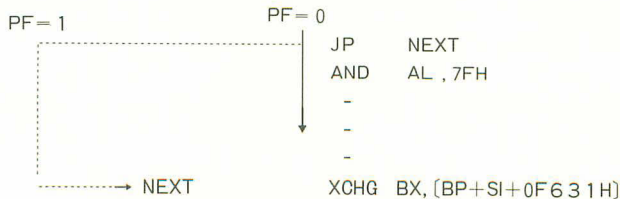
命令コードを次に示す.

次の一連の命令において,

ＪＯ命令に続いて，オーバーフロー・フラグが１ならばＸＣＨＧ命令が実行される．オーバーフロー・フラグが０ならば，ＡＮＤ命令が実行される．

サイクル数：分岐したとき：16
分岐しなかったとき：4

JP disp (Jump if Parity)
JPE disp (Jump if Parity Even)

パリティ・フラグが１ならばジャンプする．／パリティが偶数ならばジャンプする．

この命令は，パリティ・フラグが１ならばジャンプし，それ以外では次の命令を実行することを除いて，ＪＭＰ disp 命令と同じである．

命令コードを次に示す．

次の一連の命令において，

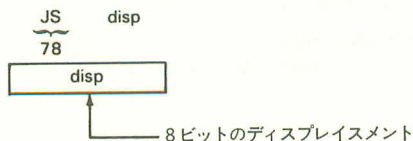
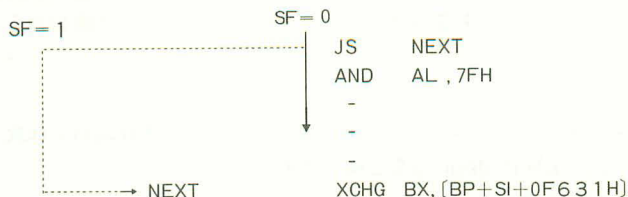
ＪＰ命令に続いて，パリティ・フラグが１ならばＸＣＨＧ命令が実行される．パリティ・フラグが０ならば，ＡＮＤ命令が実行される．

サイクル数：分岐したとき：16
分岐しなかったとき：4

JS disp (Jump if Sign)

サイン・フラグが１ならばジャンプする．

この命令は，サイン・フラグが１ならばジャンプすることを除いて，ＪＭＰ disp 命令と同じである．

命令コードを次に示す．

次の一連の命令において，

ＪＳ命令に続いて，サイン・フラグが1ならばＸＣＨＧ命令が実行される．サイン・フラグが0ならば，ＡＮＤ命令が実行される．

サイクル数：分岐したとき：16
分岐しなかったとき：4

LAHF (Load AH from 8080 Flags)

8080のフラグをＡＨレジスタにロードする．

この命令は，フラグ・レジスタの下位8ビットをＡＨレジスタに移動する．移動される8ビットを次に示す．

7	6	5	4	3	2	1	0
SF	ZF	X	AF	X	PF	X	CF

ここで，Ｘは不確定の値を表わす．

命令コードを次に示す．

LAHF
9F

例として，ＣＦとＰＦのフラグが1で，ＺＦ，ＳＦ，ＡＦのフラグが0の場合を考える．

LAHF

を実行すると，ＡＨレジスタの内容は，

00X0X1X1

になる．

LAHF
サイクル数: 4

注)

1．ステータスは影響を受けない．レジスタはAHを除いて影響を受けない．

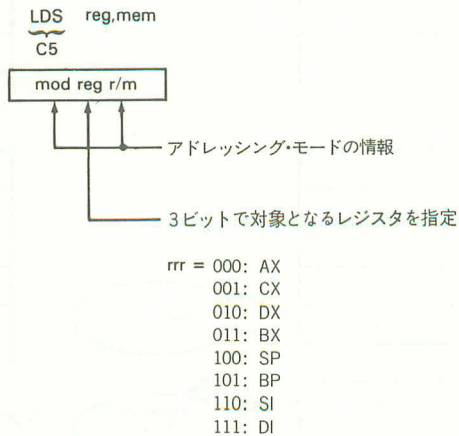
2．この命令は，8080の命令 PUSH　PSW をエミュレイトするために，PUSH　AX と共に用いられる．

8086コード	8080コード
LAHF	PUSH PSW
PUSH　AX	

LDS reg, mem　　(Load register and DS)

メモリからレジスタとＤＳにロードする．

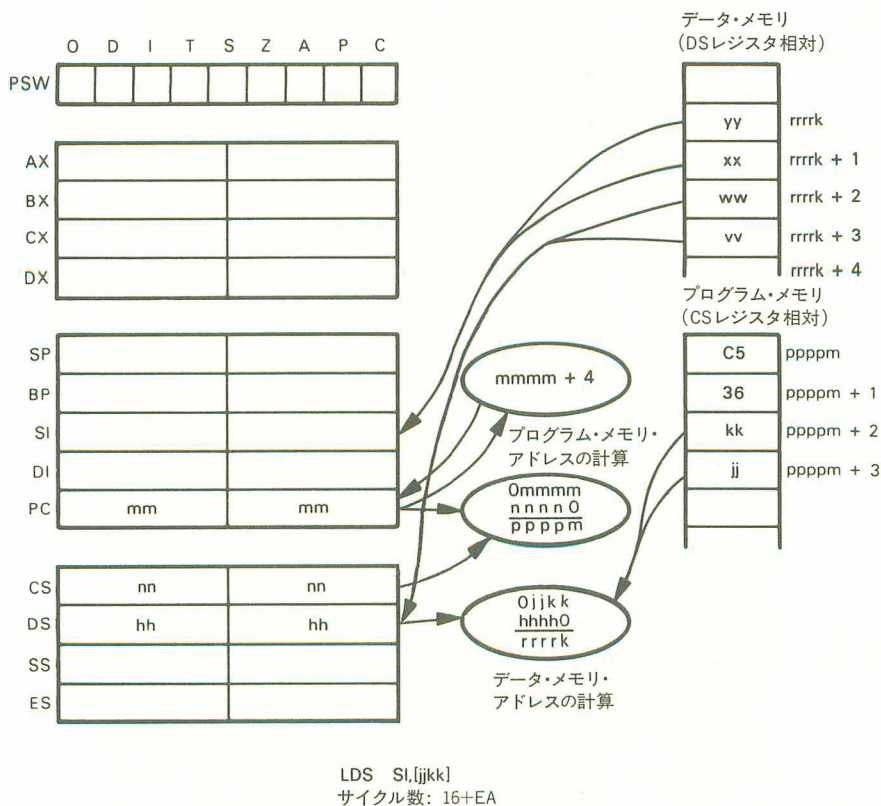
指定されたメモリ・ワードを指定されたレジスタにロードし，それに続くメモリ・ワードをＤＳレジスタにロードする．

命令コードを次に示す．

例として，ＤＳレジスタが$C000_{16}$を含み，メモリ位置$C0010_{16}$のワードが0180_{16}で，メモリ位置$C0012_{16}$のワードが2000_{16}の場合を考える．

LDS　　SI, [10H]

の実行後，ＳＩレジスタは0180_{16}に，ＤＳレジスタは2000_{16}になる．

LDS　SI,[jjkk]
サイクル数：16+EA

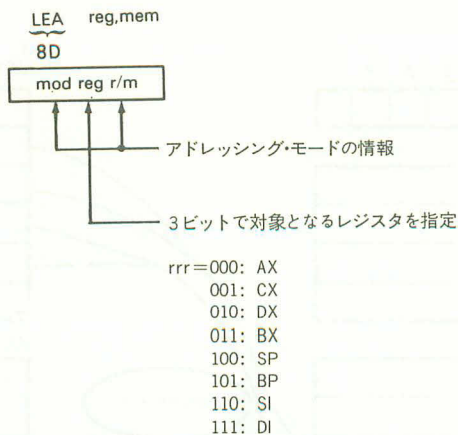
注)

1．ステータスは影響を受けない．

2．mod が11の場合，この命令による動作は定義されない．

LEA　reg，mem　　　(Load Effective Address)

オフセット・アドレスをレジスタにロードする．

メモリ・オペランドを指定する16ビットのオフセット・アドレスを，指定されたレジスタにロードする．

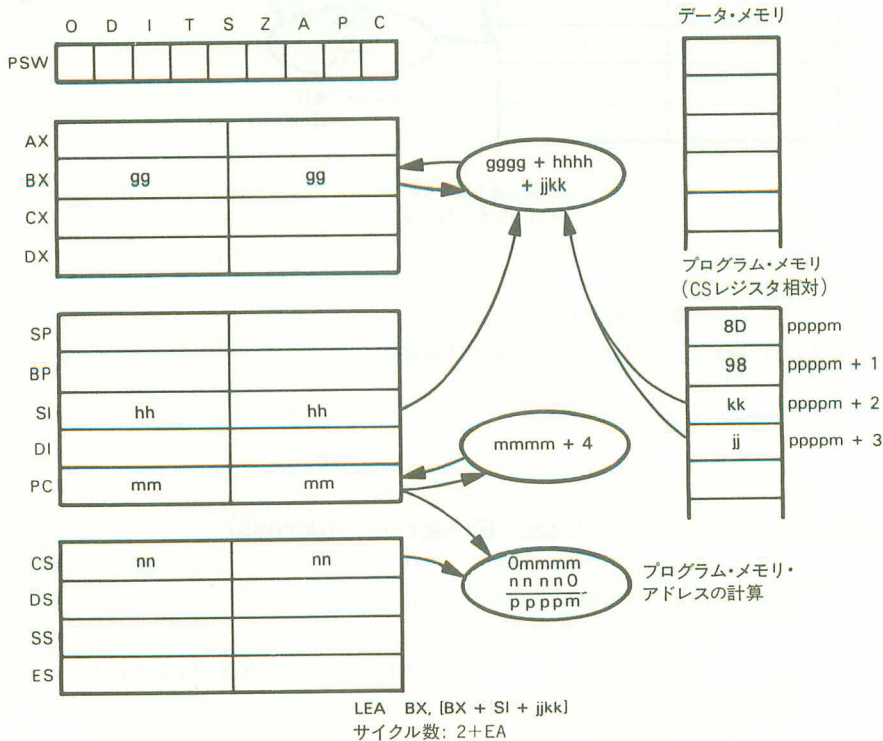
命令コードを次に示す．

BXレジスタが0400_{16}を含み，SIレジスタが$003C_{16}$を含むと仮定する．

LEA　BX,〔BX＋SI＋0F62H〕

の実行後，BXレジスタは$139E_{16}$になる．この値は，BXとSIレジスタの内容と指定されたディスプレイスメントの和である．

O D I T S Z A P C
PSW
AX
BX gg gg
CX
DX
SP
BP
SI hh hh
DI
PC mm mm
CS nn nn
DS
SS
ES
gggg + hhhh + jjkk
mmmm + 4
0mmmm
nnnn0
ppppm
プログラム・メモリ・アドレスの計算
データ・メモリ
プログラム・メモリ（CSレジスタ相対）
8D ppppm
98 ppppm + 1
kk ppppm + 2
jj ppppm + 3

LEA　BX, [BX + SI + jjkk]
サイクル数: 2+EA

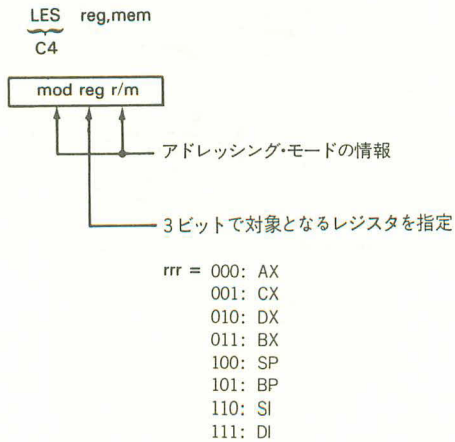
注)

1．ステータスは影響を受けない．

2．mod が11の場合，この命令による動作は定義されない．

LES reg, mem　　(Load register and ES)

メモリからレジスタとESにロードする．

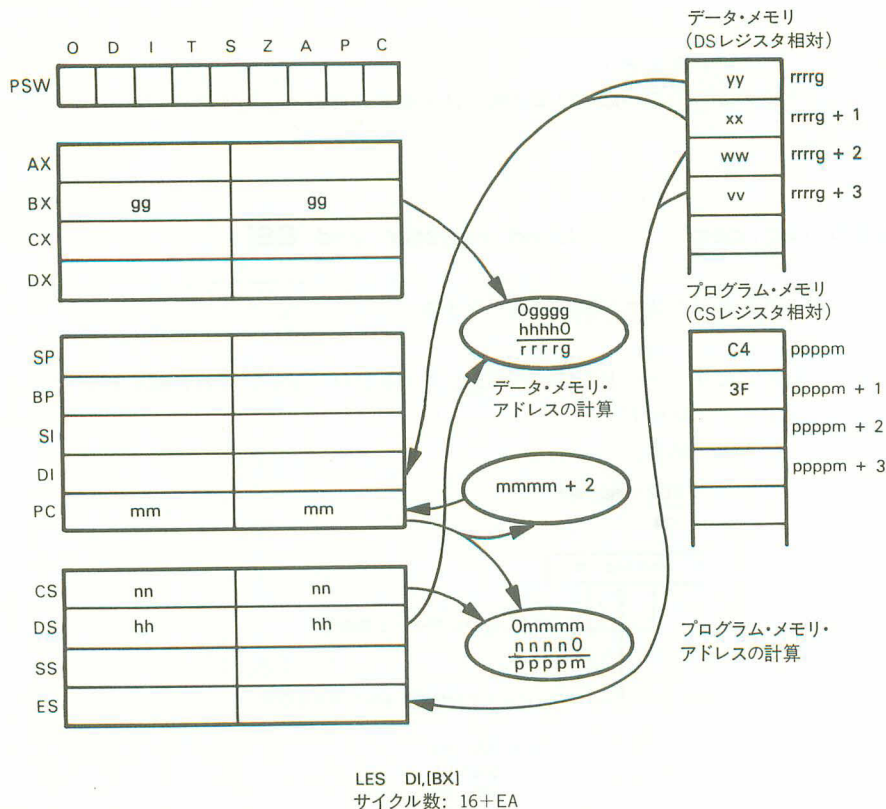
指定されたメモリ・ワードを指定されたレジスタにロードし，それに続くメモリ・ワードをESレジスタにロードする．

命令コードを次に示す．

DSレジスタが $B000_{16}$ を含み，BXレジスタが $080A_{16}$ を含み，$B080A_{16}$ におけるメモリ・ワードが $05A2_{16}$ で，$B080C_{16}$ のメモリ・ワードが 4000_{16} であると仮定する．

```
LES	DI,[BX]
```

の実行後，DIレジスタは $05A2_{16}$ に，ESレジスタは 4000_{16} になる．

LES　DI,[BX]
サイクル数：16＋EA

注)

1．ステータスは影響を受けない．

2．mod が11の場合，この命令による動作は定義されない．

3．ＤＩレジスタは本来ＥＳレジスタと関連したレジスタなので，この命令で指定されるレジスタとしてはＤＩレジスタが一般的である．

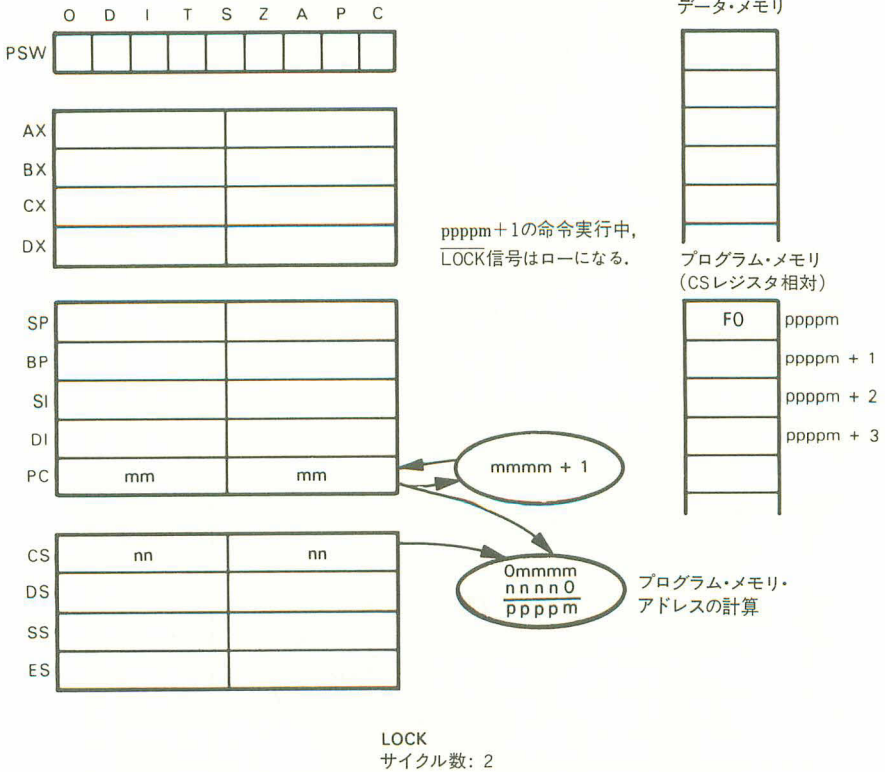
LOCK　　　(Lock)

バスのロック信号を設定する．

この命令は，8086にローの$\overline{\text{ＬＯＣＫ}}$信号を出力させるために用いられる．$\overline{\text{ＬＯＣＫ}}$信号は，次の命令実行中はローに保たれる．

この命令は，プレフィックス命令と考えられる．すなわち，$\overline{\text{ＬＯＣＫ}}$信号の設定を必要とする命令に先行して用いられる．

命令コードを次に示す．

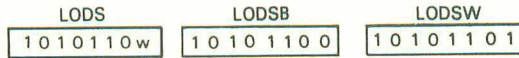
LOCK
サイクル数：2

注）

1．このプレフィックスは，任意の8086命令に前置して用いられる．しかし，このプレフィックスがＲＥＰプレフィックスとストリング・プリミティブと共に用いられた場合は問題となる．この詳細については次章を参照のこと．
2．このプレフィックスは，テスト・アンド・セットのシーケンス実行に有用である．

LODS/LODSB/LODSW　　(Load String)

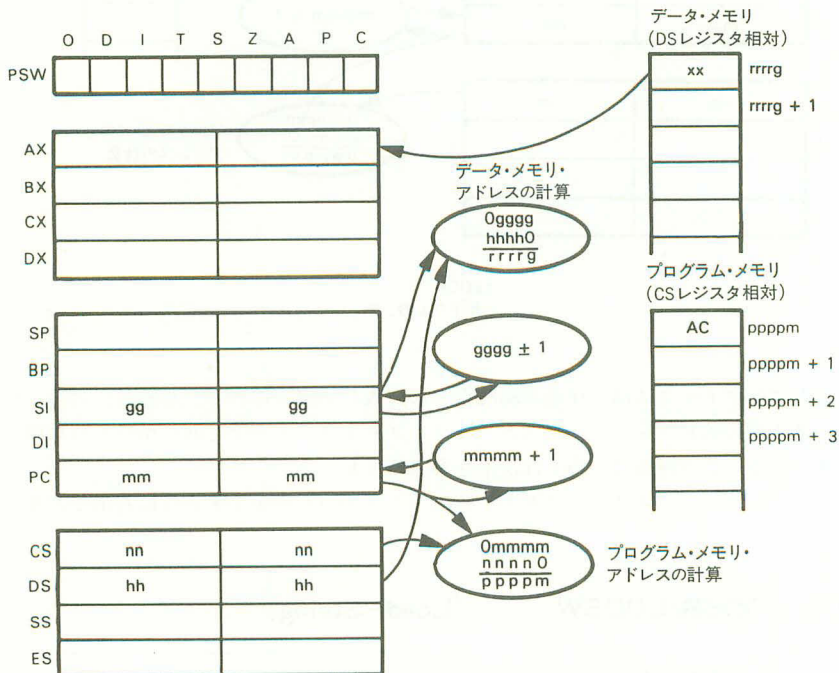
メモリからＡＬあるいはＡＸのレジスタにロードする．

ＳＩレジスタで示されるメモリの内容を，ＡＬ（８ビット操作）あるいはＡＸ（16ビッ

ト操作）のレジスタに移動する．ＳＩレジスタは，ＤＦフラグの値によって増加あるいは減少する．

命令コードを次に示す．

たとえば，ＤＦフラグが０で，ＳＩレジスタが0035_{16}を含み，ＤＳレジスタが4008_{16}を含み，メモリ位置400B5_{16}のバイトが $0F_{16}$ であると仮定する．

LODSB

の実行後，ＡＬレジスタの内容は $0F_{16}$ になり，ＳＩレジスタの内容は0036_{16}になる．

LODSB
サイクル数：12：1度だけ実行のとき
9+(13*R)：REPプレフィックスによってR回実行されたとき

注)

1．ステータスは影響を受けない．

2．デフォルト・セグメント・レジスタはＤＳレジスタである．これは，適当なセグメント変更プレフィックスによって変更される．

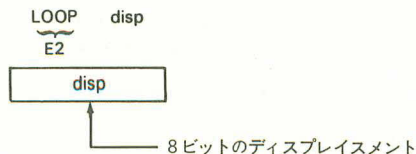
3．一般には，ＲＥＰプレフィックスはこの命令で用いられない．

4．他の8086の操作と同じく，この命令の汎用形式は，８あるいは16ビットのどちらの操作が行なわれるかを決めるために，アセンブラに何らかの記号が与えられることを必要とする．この問題についてはこの章の後で述べる．

LOOP disp　　　(Loop)

ＣＸレジスタをデクリメントし，０でなければジャンプする．

この命令は，ＣＸレジスタをデクリメントし（フラグに影響は与えない），減少によってＣＸレジスタが0にならなければジャンプを行ない，それ以外では次の命令を実行することを除けば，ＪＭＰ disp 命令と同様の機能を行なう．

命令コードを次に示す．

例として，次の一連の命令を考える．

```
                                MOV     CX, LENGTH$OFPAYROLL$ARRAY
PAYROLL$SUMMATION$ARRAY:
                                    } 給料支払い総額の計算
                                LOOP    PAYROLL$SUMMATION$ARRAY
```

PAYROLL$SUMMATION$ARRAY とＬＯＯＰ命令の間の一連の命令が，LENGTHOFPAYROLL$ARRAY 回実行される．

サイクル数：分岐したとき：17
　　　　　　分岐しなかったとき：5

LOOPZ disp　　(Loop if Zero)
LOOPE disp　　(Loop if Equal)

ＣＸレジスタをデクリメントし，ＣＸが０でなくＺＦ＝１ならば，ジャンプする．

この命令は，ＣＸレジスタをデクリメントし（フラグに影響は与えない），減少によってＣＸレジスタが０にならずしかもゼロ・フラグが１ならばジャンプを行ない，それ以外では次の命令を実行することを除けば，ＪＭＰ disp 命令と同様の機能を行なう．

命令コードを次に示す．

例として，次の一連の命令を考える．

```
        MOV     CX, NUMBER$OF$PORTS
        MOV     DX, MAIN$PORT$GROUP
        MOV     BX, REDUNDANT$PORT$GROUP
TOP:    IN      AX, DX
        INC     DX
        XCHG    BX, DX
        XCHG    AX, BP
        IN      AX, DX
        INC     DX
        XCHG    BX, DX
        CMP     AX, BP
        LOOPE   TOP
        JNZ     PORT$DISPUTE
```

ＴＯＰとＬＯＯＰＥ命令の間の一連の命令は，２つの入力ポートからのデータを比較する．１つのグループはMAIN$PORT$GROUP で示され，もう１つのグループはREDUNDANT$PORT$GROUP で示されている．次の２つの場合の１つが発生すれば，命令ＪNZ PORT$DISPUTE が実行される．

1．比較の結果，ゼロ・フラグが０に設定される．これはポートのデータが等しくない場合である．
2．ＴＯＰとＬＯＯＰＥ命令の間の命令が，NUMBEROFPORTS 回実行された場合．

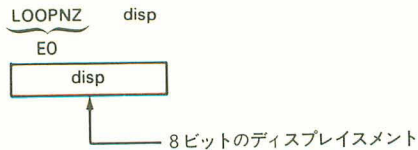
ＪＮＺ命令は，１と２の場合を区別するために用いられる．

サイクル数：分岐したとき：18
　　　　　　分岐しなかったとき：6

LOOPNZ disp　(Loop if Not Zero)
LOOPNE disp　(Loop if Not Equal)

ＣＸレジスタをデクリメントし，ＣＸが0でなくＺＦ＝0ならばジャンプする．

この命令は，ＣＸレジスタをデクリメントし（フラグに影響は与えない），減少によってＣＸレジスタが0にならずしかもゼロ・フラグが0ならばジャンプを行ない，それ以外では次の命令を実行することを除けば，ＪＭＰ disp 命令と同様の機能を行なう．

命令コードを次に示す．

例として，次の一連の命令を考える．

```
                        MOV     SI,ELEMENT$TO$MATCH
                        LES     DI
                        MOV     CX,NUMBER OF ENTRIES
SEARCH$FOR$MATCH :      -
                        -               ;SEARCH FOR MATCH
                        -
                        LOOPNE  SEARCH$FOR$MATCH
```

SEARCHFORMATCH と LOOPNE 命令の間のコードは，

1）ＣＸが減少して0になる．あるいは，

2）LOOPNE の前の命令がゼロ・フラグを1にする．たとえばマッチしたときにゼロ・フラグを1にするまで，繰り返し実行される．

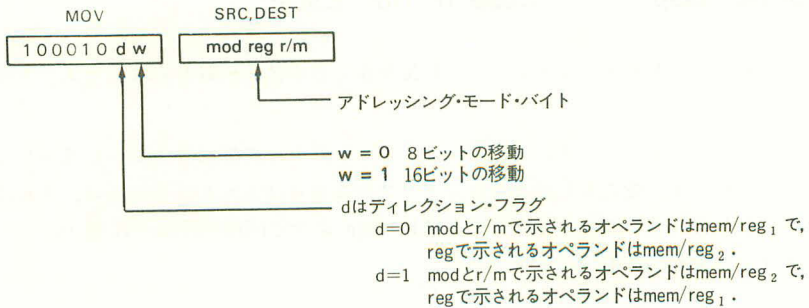
サイクル数：分岐したとき：19
　　　　　　分岐しないとき：5

MOV mem/reg1, mem/reg2　(Move)

｛レジスタからレジスタへ／メモリからレジスタへ／レジスタからメモリへ｝データを移動する．

この命令は，レジスタとレジスタあるいはメモリの間で，8あるいは16ビットのデータ

要素の移動に用いられる．

命令コードを次に示す．

たとえば，

MOV　　AX，CX

の命令は，CXレジスタの内容をAXレジスタに移動する．

O D I T S Z A P C

PSW

AX　xx　yy

BX

CX　xx　yy

DX

SP

BP

SI

DI

PC　mm　mm

CS　nn　nn

DS

SS

ES

mmmm + 2

0mmmm
nnnn0
ppppm

プログラム・メモリ・
アドレスの計算

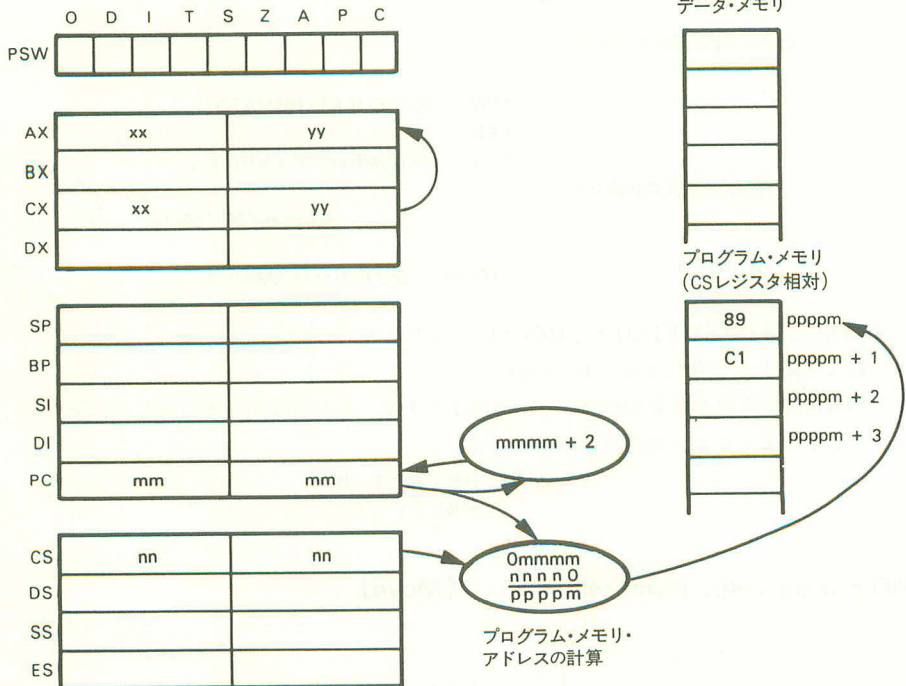
データ・メモリ

プログラム・メモリ
(CSレジスタ相対)

89　ppppm

C1　ppppm + 1

ppppm + 2

ppppm + 3

MOV　AX,CX

サイクル数:

レジスタからレジスタ:　2

メモリからレジスタ:　8+EA

レジスタからメモリ:　9+EA

注）

1．この命令では，セグメント・レジスタを指定できない．セグメント・レジスタのデータ移動については，MOV segreg, reg あるいはMOV reg ,segreg の命令を参照．

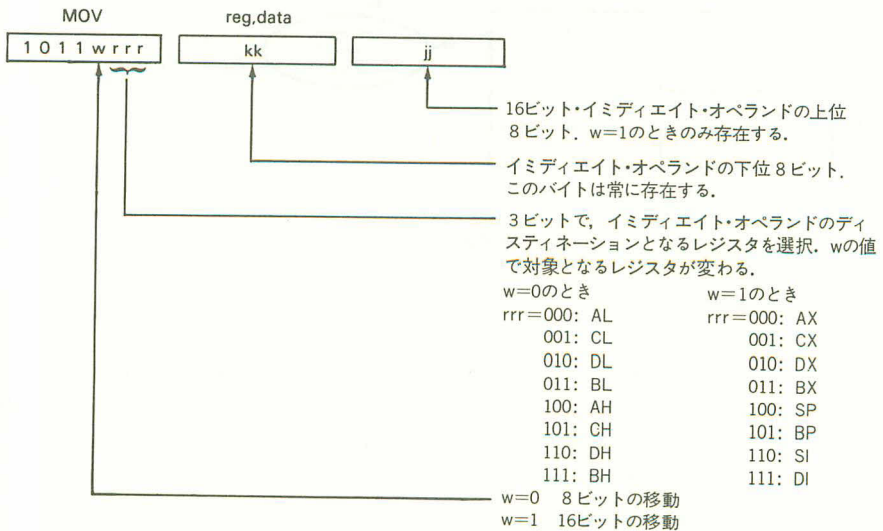
2．ステータスは影響を受けない．

3．この命令は，8080アセンブリ命令の MOV reg ,reg 命令で行なわれる機能を果たす．しかし，この命令は，対応する8080命令よりも融通性のある使い方ができる．

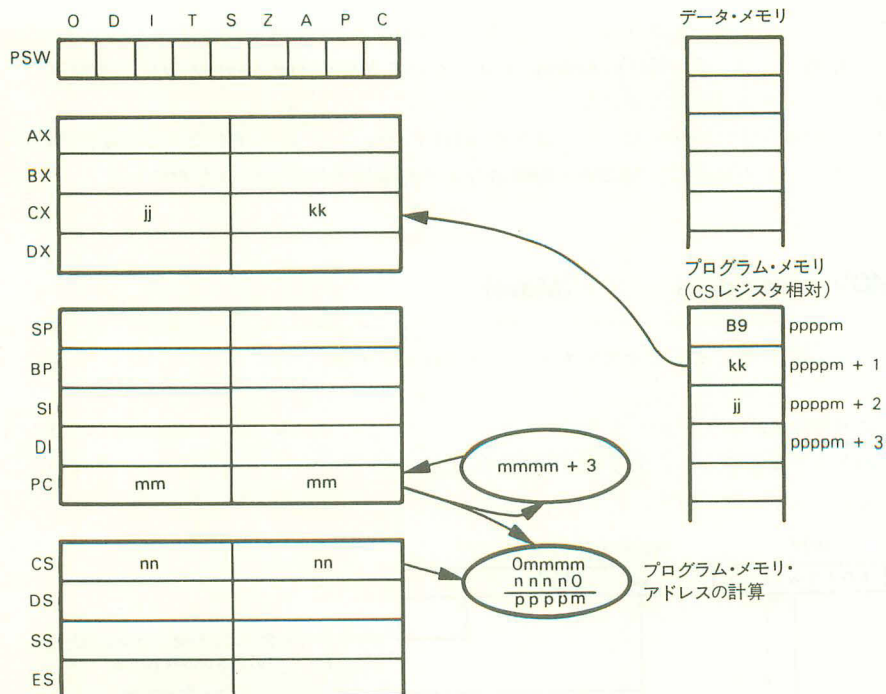
MOV reg，data　　(Move)

レジスタにイミディエイト・データをロードする．

この命令は，イミディエイト・アドレッシングによって，レジスタに８あるいは16ビットのデータ要素をロードするのに用いられる．

命令コードを次に示す．

例として，

MOV　　CX , 3168H

の命令は，ＣＸレジスタに16ビットの値 3168_{16} を移動する．

MOV　CX, jjkk
サイクル数: 4

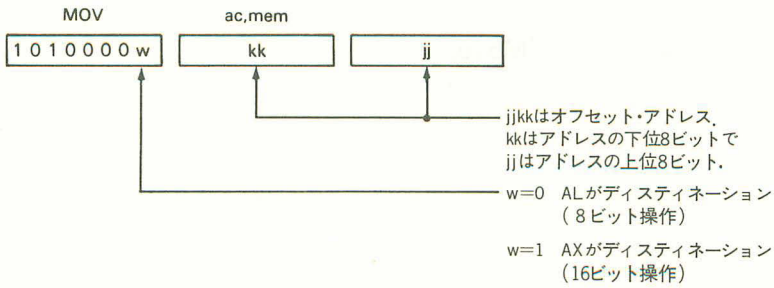
注)

1. この命令で，セグメント・レジスタにロードはできない．セグメント・レジスタにイミディエイト・データをロードするには，MOV segreg,mem/reg 命令を参照．
2. この命令は，8080で実行されるMVI（8ビットのイミディエイトの移動）とLXI（16ビットのイミディエイトの移動）の命令の機能を果たす．
3. ステータスは影響を受けない．

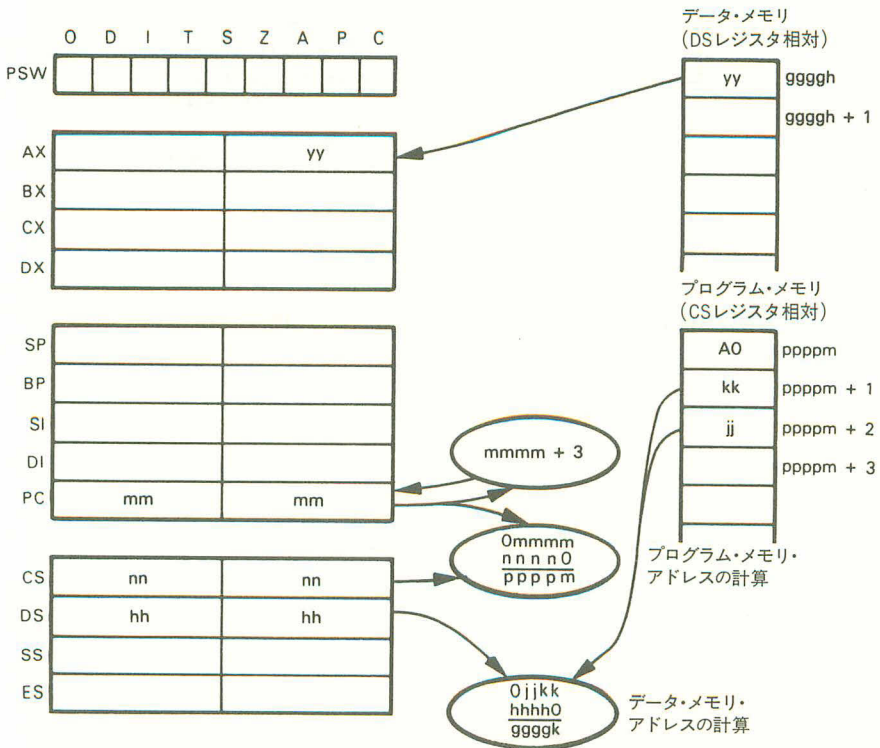
MOV　ac, mem　　　　(Move)

メモリからアキュームレータにロードする．

この命令は，メモリからアキュムレータに，8あるいは16ビットのデータ要素を移動するために用いられる．

命令コードを次に示す．

たとえば,

MOV　AL,〔1064H〕

の命令は，メモリ位置1064₁₆（DSレジスタ相対）の内容をALレジスタに移動する.

MOV　AL,[jjkk]
サイクル数：10

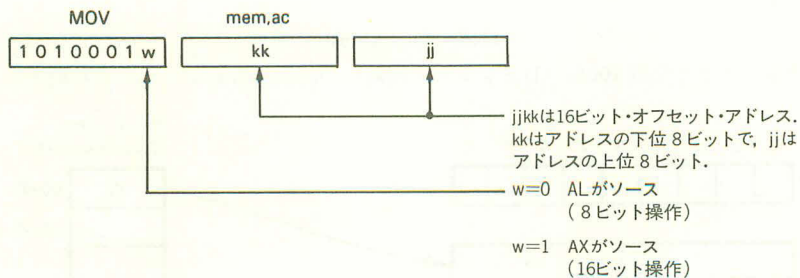
注)

1．この命令は，8080の命令　LDA addr　と同じ機能を果たす．また，AXレジスタへの16ビットのロードもできる.

MOV mem, ac (Move)

アキュムレータからメモリにストアする.

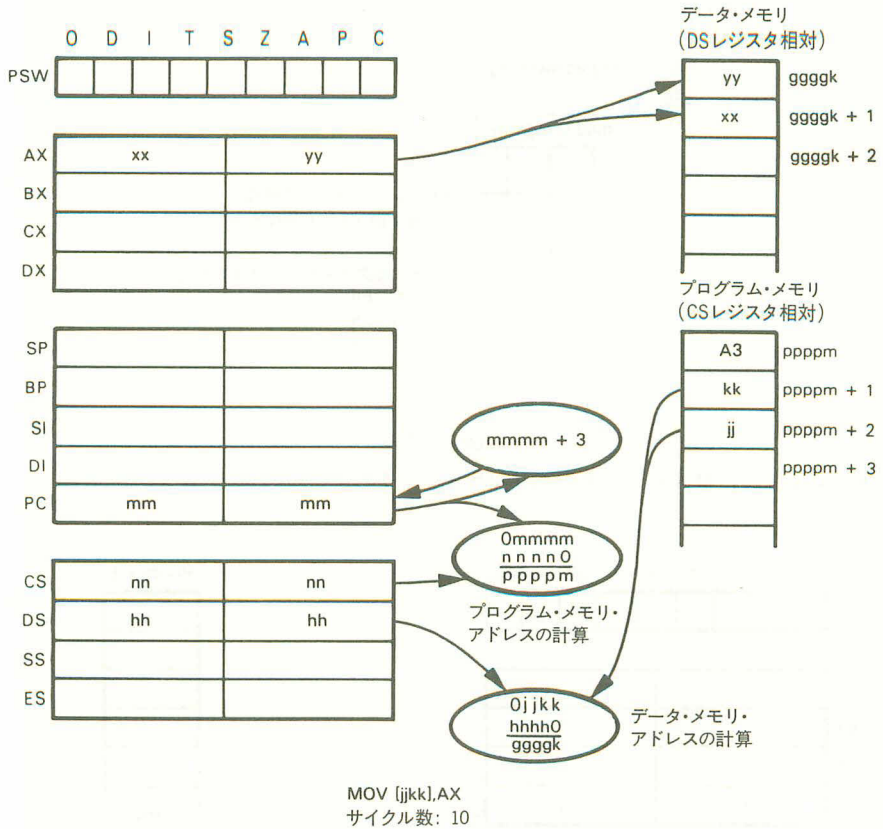
この命令はアキュムレータからメモリに，8あるいは16ビットのデータ要素を移動するために用いられる.

命令コードを次に示す.

たとえば，

MOV 〔1064H〕, AX

の命令は，AXレジスタの内容をメモリ位置 1064_{16}（DSレジスタ相対）に移動する．ALレジスタの内容は 1064_{16} に移動し，AHレジスタの内容は 1065_{16} に移動する．

MOV [jjkk],AX
サイクル数：10

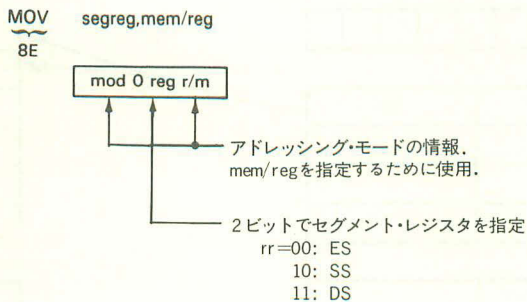
注）

1．ステータスは影響を受けない．

2．この命令は，8080の命令　STA· addr　と同じ機能を果たす．さらに，AXレジスタの16ビットのストアができる．

MOV　segreg , mem/reg　　　(Move)

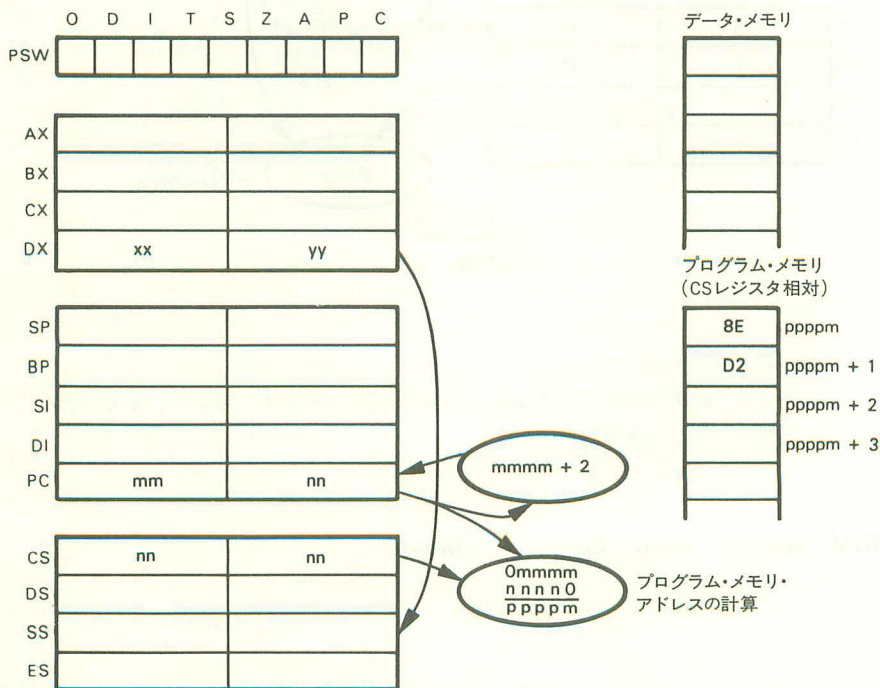
メモリあるいはレジスタのデータをセグメント・レジスタに移動する．

レジスタあるいはメモリから，16ビットのデータ要素をセグメント・レジスタに移動する．

命令コードを次に示す．

例として，

MOV　SS,DX

の命令によって，ＤＸレジスタの内容がＳＳレジスタに移動する．

O D I T S Z A P C
PSW
AX
BX
CX
DX　xx　yy
SP
BP
SI
DI
PC　mm　nn
CS　nn　nn
DS
SS
ES
mmmm + 2
0mmmm
nnnn0
ppppm
プログラム・メモリ・
アドレスの計算
データ・メモリ
プログラム・メモリ
（CSレジスタ相対）
8E　ppppm
D2　ppppm + 1
ppppm + 2
ppppm + 3

MOV　SS,DX
サイクル数：レジスタからレジスタ：2
　　　　　　メモリからレジスタ：　8+EA

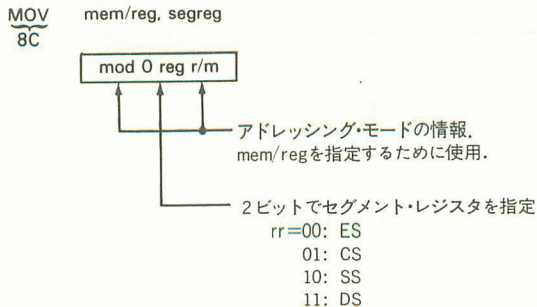
注)

1．reg＝01の場合，この命令の結果は定義されない．この制限によって，ユーザがＣＳレジスタに直接ストアすることを防止している．ＣＳレジスタに対する変更は，ＰＣへのロードも行なうＪＭＰ，ＣＡＬＬ，ＲＥＴ，ＩＲＥＴ，ＩＮＴの命令によってのみ行なわれる．
2．この命令は，一般にプログラムのセグメント領域が定義される初期設定で用いられる．
3．インタラプトは，この命令の終わりではサンプルされない．これに続く命令の終わりにサンプルされる．この制限は，インタラプトを起こさせずに，32ビットのポインタ全体の回復を可能にする．これは，ＳＳとＳＰをロードする際に問題となる．

MOV　mem/reg, segreg　　(Move)

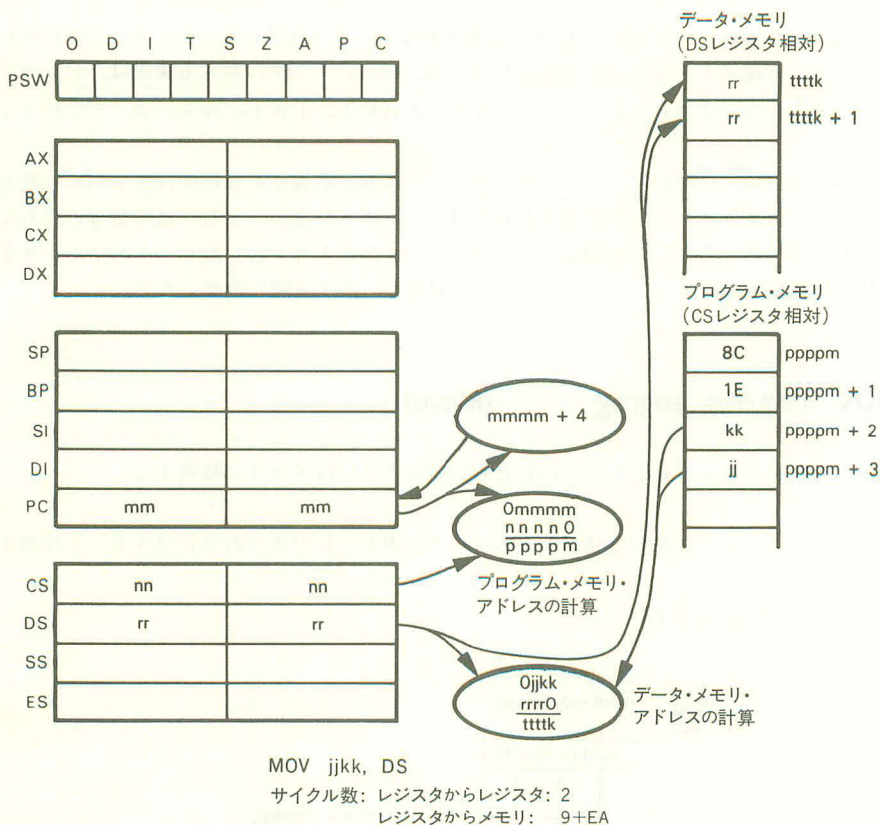
セグメント・レジスタの内容をレジスタあるいはメモリに移動する．

セグメント・レジスタから16ビットのデータ要素を，レジスタあるいはメモリに移動する．

命令コードを次に示す．

たとえば，ＤＳレジスタが2000_{16}含む場合を考える．

MOV　2000H, DS

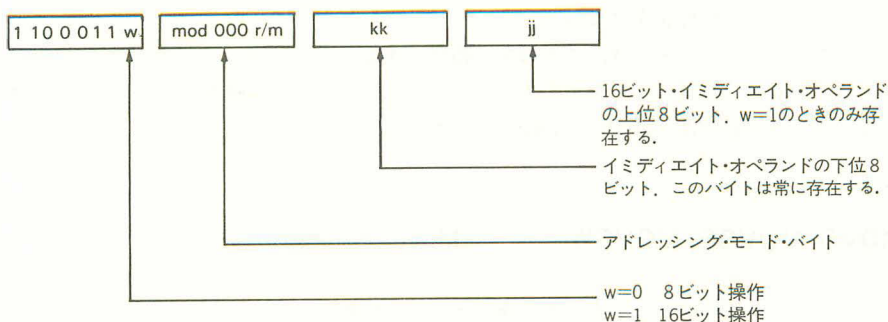
の命令によって，メモリの22000_{16}に00_{16}がストアされ，22001_{16}に20_{16}がストアされる．

注)

1．これは，汎用レジスタ間のＭＯＶではなく，セグメント・レジスタの移動用である．
汎用レジスタのＭＯＶについては，MOV mem/reg 1,mem/reg 2 を参照．

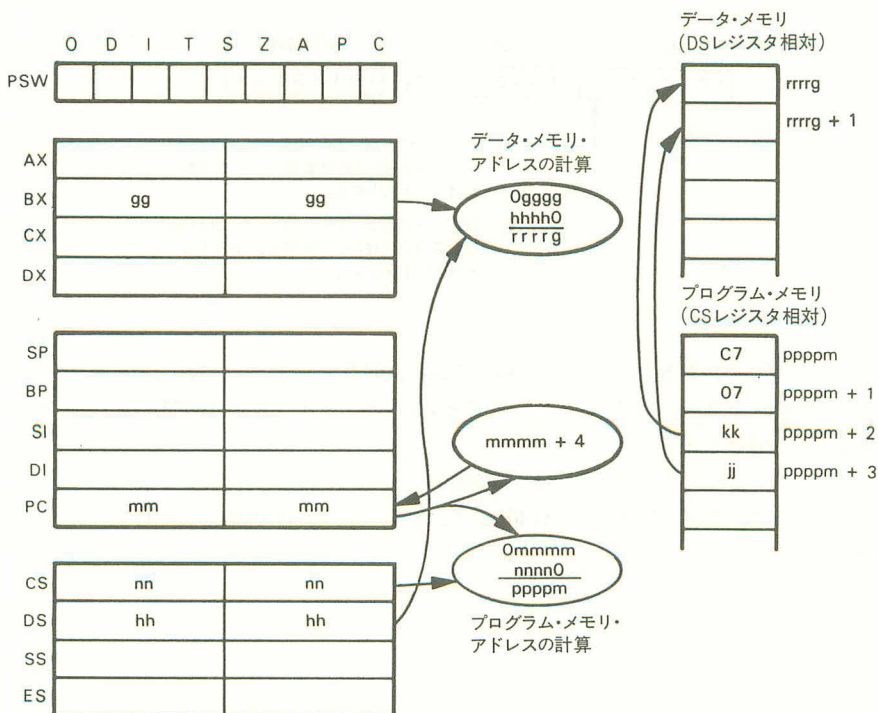
MOV mem/reg, data (Move)

イミディエイト・データをレジスタあるいはメモリに移動する．

オペ・コードに続くバイトのイミディエイト・データを，指定されたレジスタあるいはメモリ位置に移動する．

命令コードを次に示す．

たとえば，DSレジスタがD000₁₆を含み，BXレジスタが0016₁₆を含む場合を考える．

MOV　〔BX〕, 491FH

の実行後，メモリ位置D0016₁₆は1F₁₆に，メモリ位置D0017₁₆は49₁₆になる．

MOV　[BX],jjkk
サイクル数：10+EA

注）

1．ステータスは影響を受けない．

2．セグメント・レジスタは，この命令で指定できない．

3．この命令は一般に，イミディエイト・データをレジスタに移動するためには用いられない．そのためには，MOV reg,data の命令がある．

MOVS/MOVSB/MOVSW　　　(Move String)

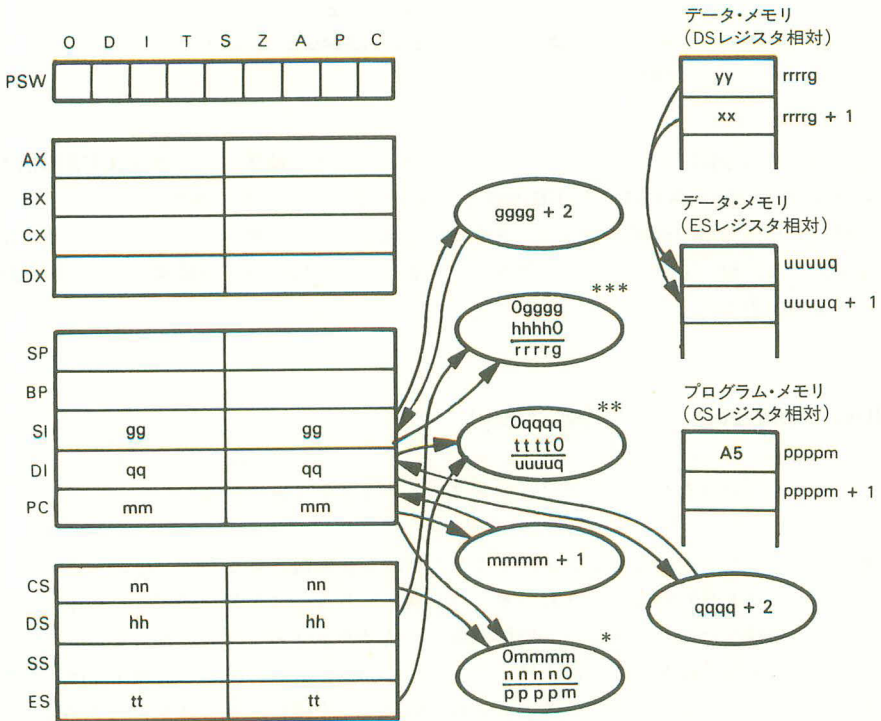
バイトあるいはワードをメモリからメモリへ移動する．

8あるいは16ビットを，ＳＩレジスタで示されるメモリ位置からＤＩレジスタで示されるメモリ位置に移動する．ＳＩとＤＩのレジスタは，ＤＦフラグの値によって，増加あるいは減少する．

命令コードを次に示す．

ＤＦフラグが0で，ＤＳレジスタが1000_{16}を含み，ＥＳレジスタが1780_{16}を含み，ＳＩレジスタが0006_{16}を含み，ＤＩレジスタが0006_{16}を含み，メモリ位置10006_{16}のワードが$8F0B_{16}$である場合を考える．

MOVSW

の実行後，メモリ位置17806_{16}は $8F0B_{16}$ で，ＳＩレジスタは0008_{16}に，ＤＩレジスタは0008_{16}になる．

* プログラム・メモリ・アドレスの計算
　** ディスティネーション・データ・メモリ・アドレスの計算
*** ソース・データ・メモリ・アドレスの計算

MOVSW

サイクル数：18：1度だけ実行のとき
　　　　　　9+(17*R)：REPプレフィックスによってR回実行されたとき

注)

1．ステータスは影響を受けない．

2．ソース・オペランドのデフォルト・セグメント・レジスタはＤＳレジスタである．このセグメントは，セグメント・プレフィックスを用いて変えられる．ディスティネーション・オペランドのデフォルト・セグメント・レジスタはＥＳレジスタである．このセグメントは，セグメント・プレフィックスによって変えられない．

3．ＲＥＰプレフィックスやＬＯＣＫプレフィックスはこの命令と共に用いられる．この命令と共に，ＲＥＰとＬＯＣＫのプレフィックスが用いられた場合は問題となる．この問題については次章を参照．

4．この命令は，メモリ・ブロックの移動に便利である．次の一連の命令を考える．

```
LES     DI,CURRENT$START$CF$PRINT$BUFFER
MOV     SI, PAGE$HEADER$MESSAGE
MOV     CX, PAGE$HEADER$MESSAGE$LENGTH
REP
MOVS    BYTE
```

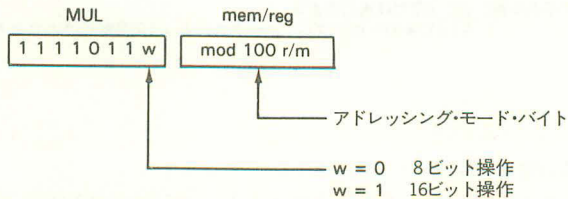
これは，PAGE$HEADER$MESSAGEで示されるメモリ位置のデータを，CURRENT$START$OF$PRINT$BUFFERの内容で示されるメモリ位置に移動する．

5．MOVSの汎用の形式に対して，8あるいは16ビットの移動のどちらをどのように指定するかは，用いるアセンブラに依存している．この件についての解説は，この章の終わりを参照のこと．

MUL mem/reg (Multiply)

ALあるいはAXレジスタにレジスタあるいはメモリを乗じる．

2つのオペランドを符号なし数値，すなわち単純な2進数と見なして，指定されたレジスタあるいはメモリの内容と，AL（8ビット操作）あるいはAX（16ビット操作）のレジスタとの乗算を行なう．8ビット操作の場合．結果の下位8ビットはALレジスタにストアされ，結果の上位8ビットはAHレジスタにストアされる．16ビット操作では，結果の下位16ビットはAXレジスタにストアされ，結果の上位16ビットはDXレジスタにストアされる．どちらの場合も，結果の上位½が0ならば，OFとCFは0となり，そうでなければAHあるいはDXに有効数字があることを示すために，OFとCFは1になる．

命令コードを次に示す．

例として，AXレジスタが4514_{16}を含み，CLレジスタが05_{16}を含む場合を考える．

```
MUL     AL,CL
```

の実行後，AXレジスタは0064_{16}になり，キャリーとオーバーフローのフラグは0となる．

MUL　AL,CL

サイクル数：8ビットのメモリによる乗算：(76－83) ＋EA
　　　　　　16ビットのメモリによる乗算：(124－139) ＋EA
　　　　　　8ビットのレジスタによる乗算：70－77
　　　　　　16ビットのレジスタによる乗算：118－133

注)

1．これは符号なしの乗算である．2つのオペランドは，以下の範囲の符号なし2進数として取り扱われる．

　8ビット：0～255

　16ビット：0～65535

　符号付きの乗算については，命令ＩＭＵＬを参照．

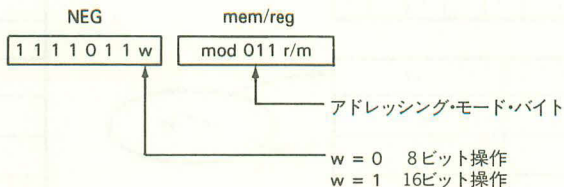
2．ある場合には，乗算を行なうためにシフトを用いることがより適当となるときがある．このような場合は，メモリの保存が最も重要ではなく速さが必要なときに生じる．

NEG mem/reg　　　(Negate)

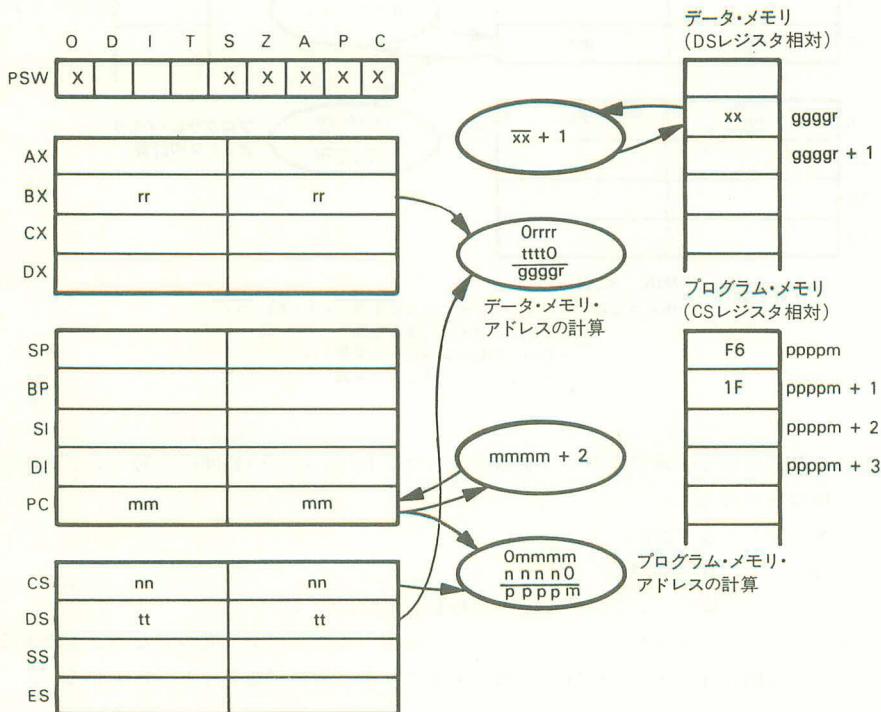
レジスタあるいはメモリの内容の補数をとる．

この命令は，0から指定されたオペランドの2の補数による減算を行なう．結果は指定

されたオペランドにストアされる．8あるいは16ビットのオペランドが指定できる．

命令コードを次に示す．

BXレジスタが0006_{16}を含み，DSレジスタが1800_{16}を含み，メモリ位置18006_{16}の内容が47_{16}であるとする．

NEG 〔BX〕

の命令実行後，メモリ位置18006_{16}の内容はB 9_{16}になる．

NEG [BX]
サイクル数：メモリ･オペランド：16＋EA
レジスタ･オペランド：3

注）

1．8080アセンブリ言語には，等価な命令は存在しない．16ビットの数値に対してこの命

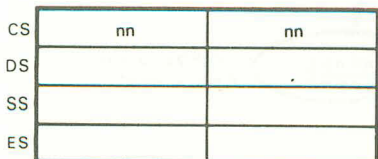
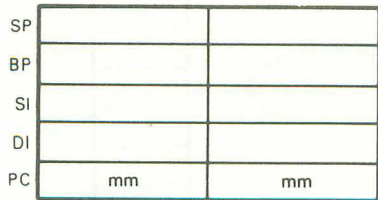
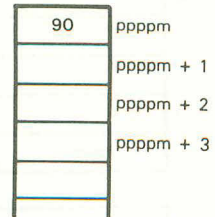
令と等価な8080の処理は次のようになる．

```
MOV     A, D
CMA
MOV     D, A
MOV     A, E
CMA
MOV     E, A
INX     D
```

NOP　　(No Operation)

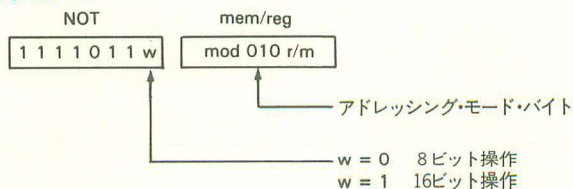
無操作（ノー・オペレーション）．

何も実行しない．命令コードを次に示す．

NOP
90

NOP
サイクル数：3

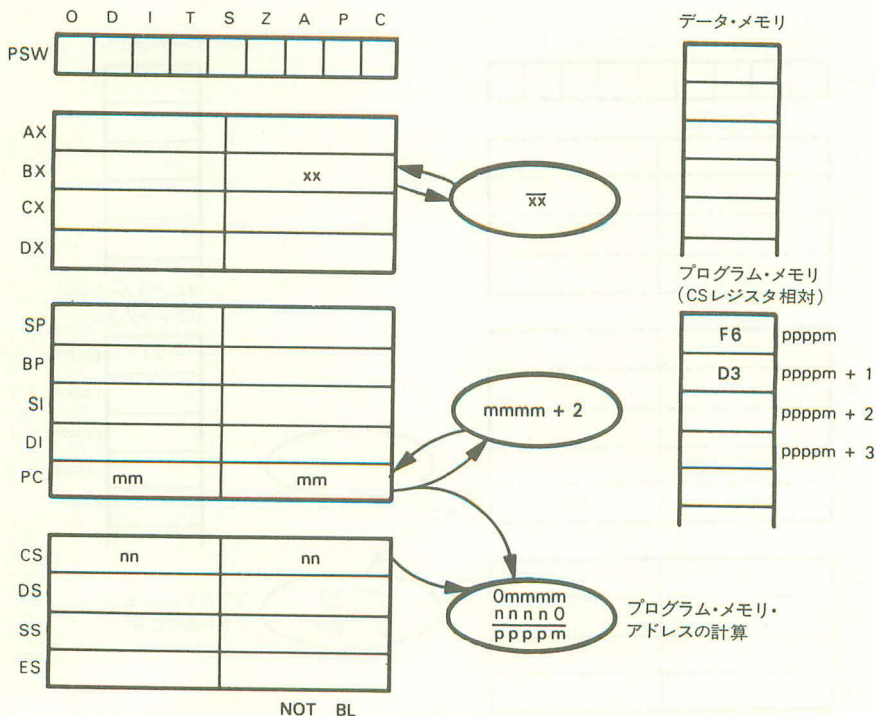
NOT mem/reg (NOT)

レジスタあるいはメモリの1の補数をとる．

指定されたレジスタあるいはメモリ位置の内容の補数をとる．
命令コードを次に示す．

ＢＬレジスタが FB_{16} を含むとする．

NOT　BL

の実行後，ＢＬレジスタは 04_{16} になる．

NOT　BL

サイクル数：メモリ・オペランド：16＋EA
　　　　　　レジスタ・オペランド：3

注)

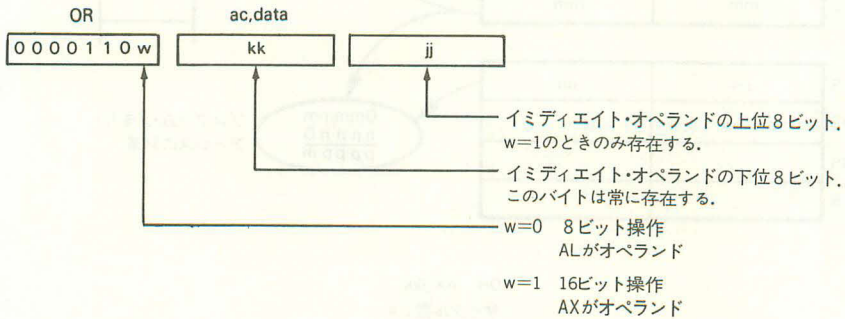
1．ステータスは影響を受けない．

2．この命令は，8080の命令　CMA　と同じ機能を果たす．この命令は16ビットの補数をとることもでき，任意の汎用レジスタあるいはメモリ位置も指定できる．

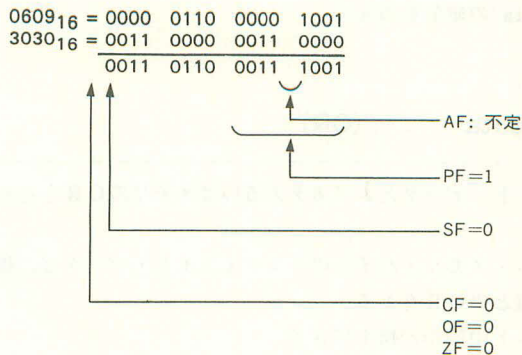
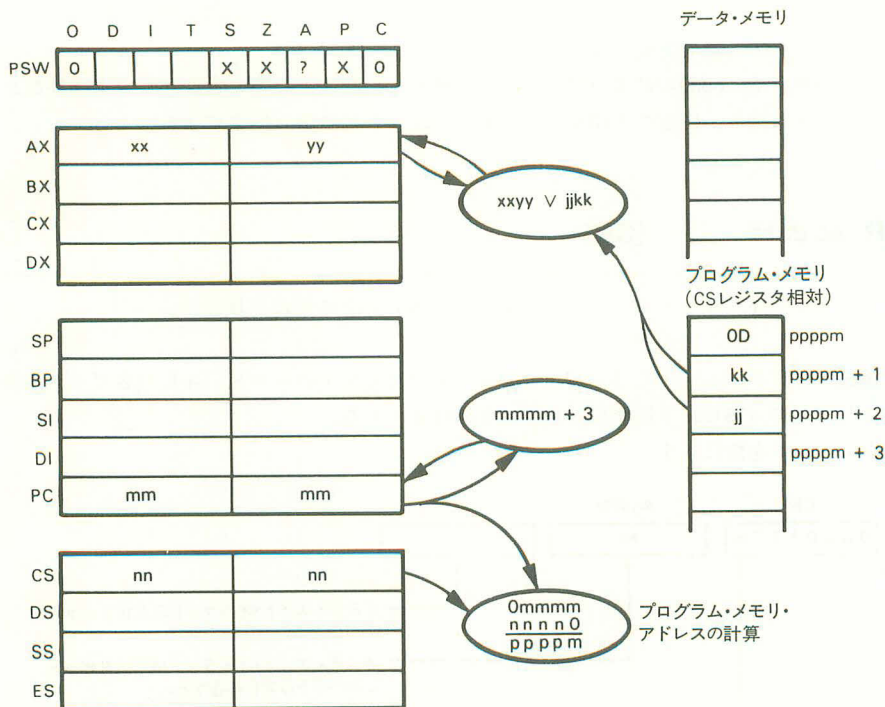
OR　ac,data　　　　(OR)

イミディエイト・データとAXあるいはALレジスタのORをとる．

後続のプログラム・メモリ・バイトのイミディエイト・データと，AL（8ビット操作）あるいはAX（16ビット操作）のレジスタのORをとる．

命令コードを次に示す．

AXレジスタが0609_{16}を含むと仮定する．

OR　AX,3030H

の実行後，AXレジスタは3639_{16}になる．

OR　AX,jjkk
サイクル数: 4

注)

1. この命令は，8080の命令　ＯＲＩ　data　と同じ機能を果たす．また，16ビットの操作を行なうこともできる．

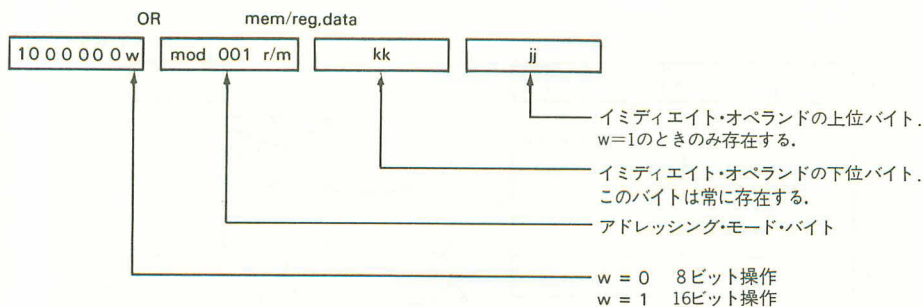
2. イミディエイト・データと，他の汎用レジスタあるいはメモリとのＯＲが必要ならば，OR mem/reg, data の命令を参照．

OR mem/reg, data　　　(OR)

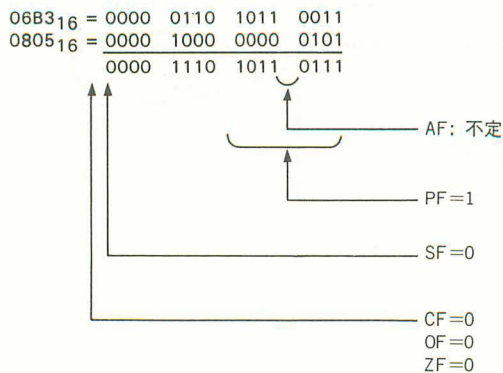
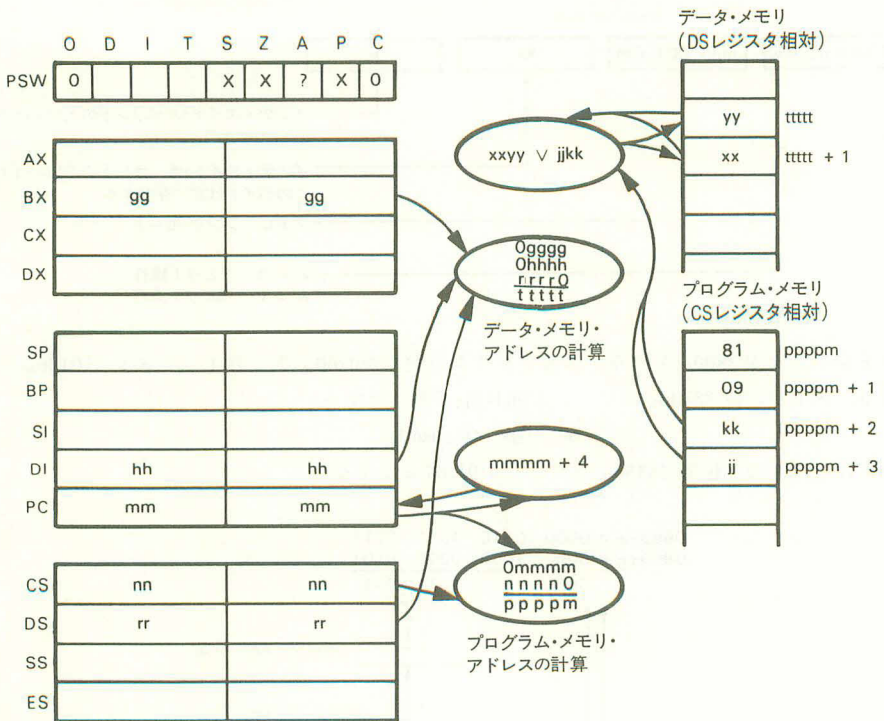
イミディエイト・データとレジスタあるいはメモリのＯＲをとる．

後続のプログラム・メモリ・バイトのイミディエイト・データと，指定されたレジスタあるいはメモリ位置とのＯＲをとる．

8あるいは16ビットの操作が指定できる．

命令コードを次に示す．

DSレジスタが 3800_{16} を含み，BXレジスタの内容が 0200_{16} で，DIレジスタが 0136_{16} を含み，メモリ位置 38336_{16} のワードが $06B3_{16}$ であるとする．

OR　〔BX+DI〕, 0805H

の実行後，メモリ位置 38336_{16} のワードは $0EB7_{16}$ になる．

注)

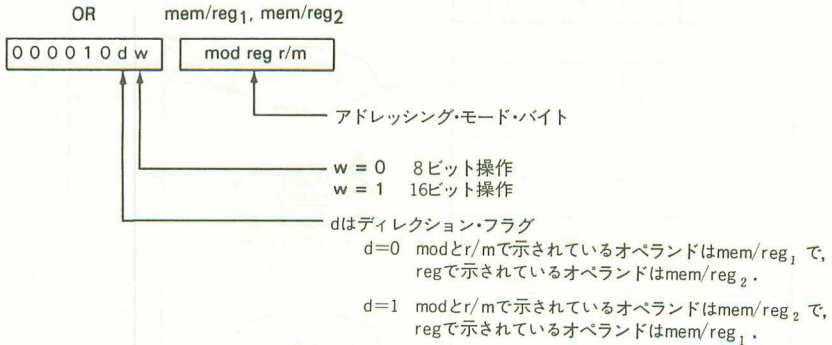
1．この命令は，一般にイミディエイト・データとＡＸあるいはＡＬレジスタのＯＲをとるためには用いられない．このためには，ＯＲ ac,data の命令がある．

OR mem/reg₁,mem/reg₂ (OR)

$\left\{ \begin{array}{l} \text{レジスタとレジスタ} \\ \text{レジスタとメモリ} \\ \text{メモリとレジスタ} \end{array} \right\}$ のＯＲをとる．

mem/reg₂で示されるレジスタあるいはメモリ位置の内容と，mem/reg₁で示されるレジスタあるいはメモリ位置の内容のＯＲをとり，結果を mem/reg₁ に返す．8あるいは16ビットの操作が指定できる．mem/reg₁ あるいはmem/reg₂はメモリ・オペランドとなるが，オ

ペランドの一方はレジスタ・オペランドでなければならない．

命令コードを次に示す．

AXレジスタが0060_{16}を含み，DSレジスタが4000_{16}を含み，BXレジスタが$009A_{16}$を含み，メモリ位置$4009A_{16}$のワードが$012C_{16}$であるとする．

OR　〔BX〕, AX

の実行後，メモリ位置$4009A_{16}$のワードは$016C_{16}$になる．

フラグは次のように設定される．

データ・メモリ
（DSレジスタ相対）

O D I T S Z A P C

PSW 0 X X ? X 0

AX xx yy
BX gg gg
CX
DX

xxyy ∨ vvww

ww rrrrg
vv rrrrg + 1
rrrrg + 2

0gggg
hhhh0
rrrrg

データ・メモリ・
アドレスの計算

プログラム・メモリ
（CSレジスタ相対）

09 ppppm
07 ppppm + 1
ppppm + 2
ppppm + 3

SP
BP
SI
DI
PC mm mm

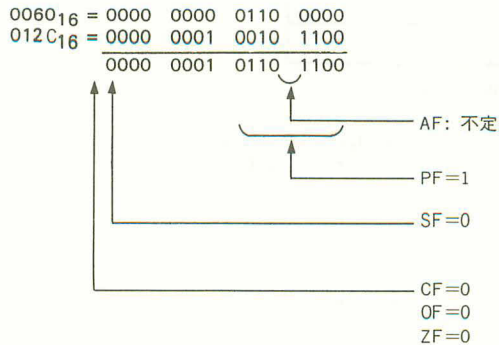
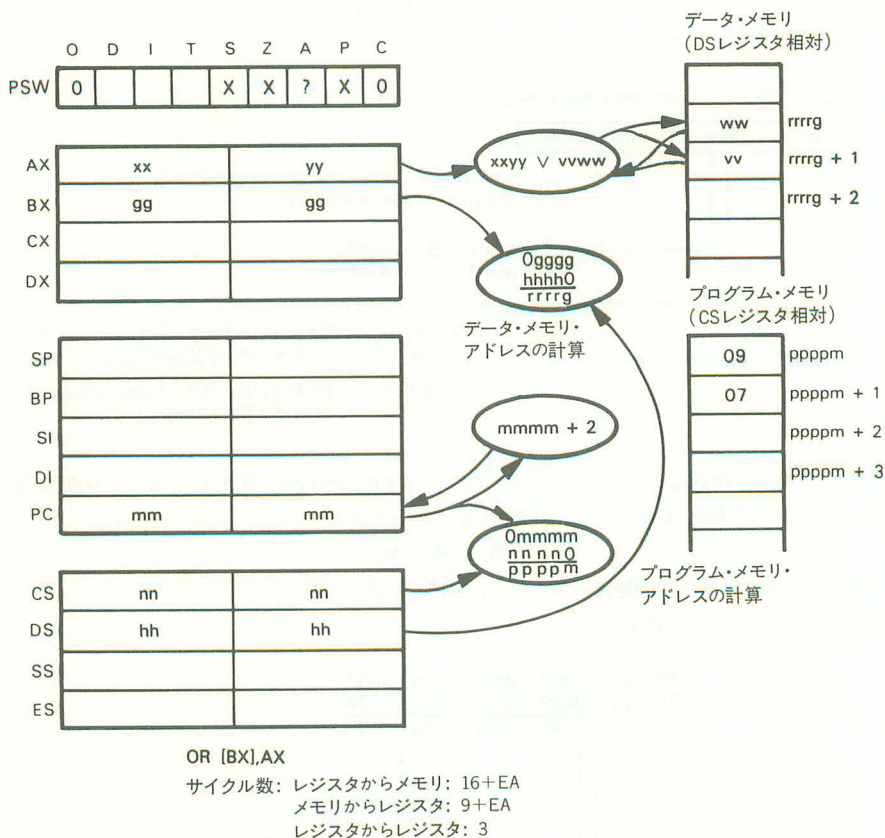
mmmm + 2

0mmmm
nnnn0
ppppm

プログラム・メモリ・
アドレスの計算

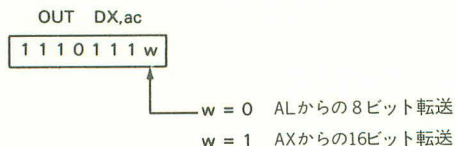
CS nn nn
DS hh hh
SS
ES

OR [BX],AX

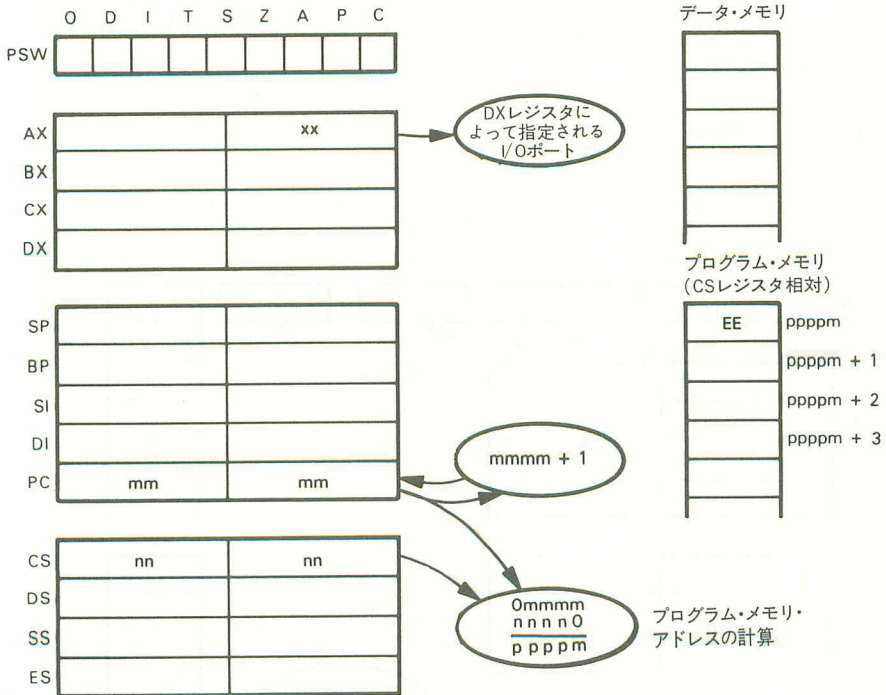
サイクル数：レジスタからメモリ：16＋EA
メモリからレジスタ：9＋EA
レジスタからレジスタ：3

OUT DX,ac (Output)

アキュムレータの内容を出力する．

AL（8ビット）あるいはAX（16ビット）のレジスタの8あるいは16ビットのデータ要素を，DXレジスタの内容で指定されるI/Oポートに出力する．

命令コードを次に示す．

例として，ＤＸレジスタがFFF2$_{16}$を含み，ＡＬレジスタが40_{16}の場合を考える．

OUT　DX,AL

の実行によって，ポート番号FFF2$_{16}$のI/Oポートのバッファに，40_{16}がロードされる．

OUT　DX,AL
サイクル数：8

注）

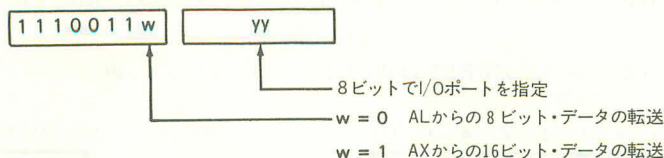
1．この命令によって，0からFFFF$_{16}$までの番号で指定されるI/Oポートにアクセスできる．

2．レジスタあるいはステータスは影響を受けない．

OUT　port,ac　　　(Output)

アキュムレータの内容を出力する．

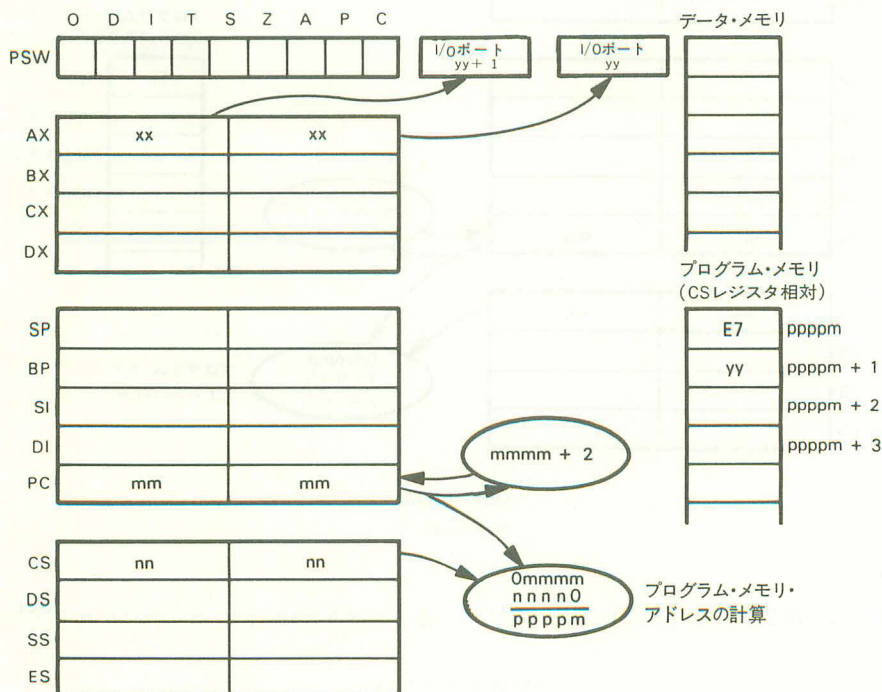
この命令は，ＡＬ（８ビット）あるいはＡＸ（16ビット）のレジスタの８あるいは16ビットのデータ要素を，命令の２番目のバイトで指定されるＩ/Ｏポートに出力する．

命令コードを次に示す．

レジスタあるいはフラグは影響を受けない.

AXレジスタが58A4$_{16}$を含むとする.

OUT　14H,AX

の実行によって，14$_{16}$のI/Oポートに A4$_{16}$が，15$_{16}$のI/Oポートに58$_{16}$が転送される.

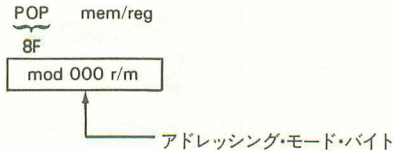
OUT　yy,AX
サイクル数: 10

注)

1. この命令によって，0からFF$_{16}$までの番号で指定されるI/Oポートにアクセスできる．この範囲外のポートについては，OUT DX,ac の命令を参照.
2. この命令は，8080の命令 OUT port と同じ機能を果たす．さらに1個の命令で16ビット・データの転送が可能である（8080の命令 OUT port では不可能）.

3．ＯＵＴ命令を有効に用いるためには，ハードウエアの構成を十分に理解することが必要である．I/Oロジックの構成方法によって，種々のハードウエアの機能にアクセスする際に用いられるポート・アドレスが決定される．特定のメモリ・アドレスでメモリ参照命令を用いることで外部ロジックにアクセスするマイクロコンピュータ・システムを設計することもできる*.

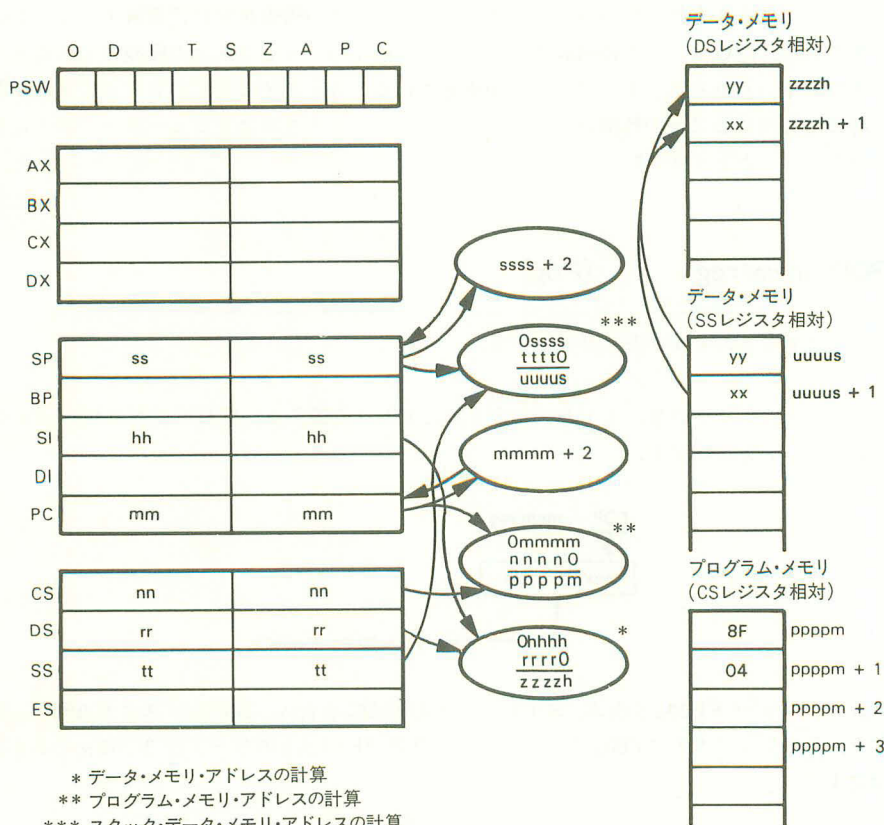
POP　mem/reg　　　　(Pop)

スタックのトップからリードする．

スタックのトップの２バイトを，指定されたメモリ位置あるいはレジスタにポップする．命令コードを次に示す．

ＤＳレジスタが FF00$_{16}$ を含み，ＳＩレジスタが 0008$_{16}$ を含み，ＳＰレジスタが 0FEA$_{16}$ を含み，ＳＳレジスタが 2F00$_{16}$ を含み，メモリ位置 2FFEA$_{16}$ のワードが 3CB5$_{16}$ であると仮定する．

POP　〔SI〕

の実行後，メモリ位置FF008$_{16}$ の内容はC5$_{16}$ になり，メモリ位置FF009$_{16}$ の内容は3B$_{16}$ になる．ＳＰは0FFC$_{16}$ になる．

＊　メモリ・マップドI/O　(Memory mapped I/O)（訳者注）.

POP [SI]

サイクル数: メモリ・オペランド: 17+EA
レジスタ・オペランド: 8

注)

1. この命令は，一般にデータをレジスタにポップするためには用いられない．命令 POP reg はこの機能を果たし，プログラム・メモリの1バイトしか占有しない．
2. ステータスは影響を受けない．

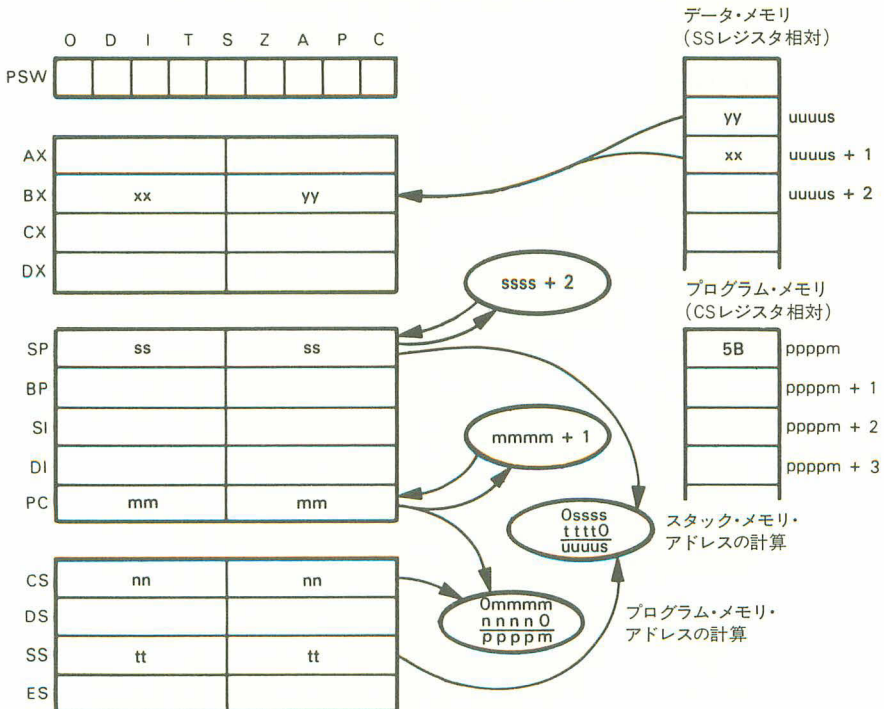
POP reg (Pop)

スタックのトップからリードする．

スタックのトップの2バイトを，指定されたレジスタにポップする．
命令コードを次に示す．

たとえば，次の命令を考える．

POP　BX

この命令は，スタック・ポインタ（スタック・セグメント）で示されるバイトをBLにポップし，スタック・ポインタをインクリメントしてそのとき示されるバイトをBHにポップする．最後に，再びスタック・ポインタを1だけ増加して，スタックの新しいトップを示すようにする．これで，実際8086における1つの16ビット転送が行なわれる．

POP　BX
サイクル数：8

注)

1．この命令は，セグメント・レジスタへのデータ要素のポップには使用できない．セグメント・レジスタへデータをポップするためには，命令 ＰＯＰ segreg を参照．

2．この命令が意味を持つためには，当然以下のことが必要となる．

a.　スタック・ポインタが初期設定されている．

b.　ＰＵＳＨ命令によってスタックにデータが存在する．

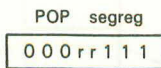
当然，ＳＰレジスタを２だけ増加する目的でこの命令を用いることができるが，これは推奨できない．

3．この命令は，8080アセンブリ言語の命令 ＰＯＰ reg と同じ機能を果たす．

POP segreg　　(Pop)

スタックのトップからリードする．

この命令は，スタックのトップから２バイトを，指定された16ビットのセグメント・レジスタにポップする．

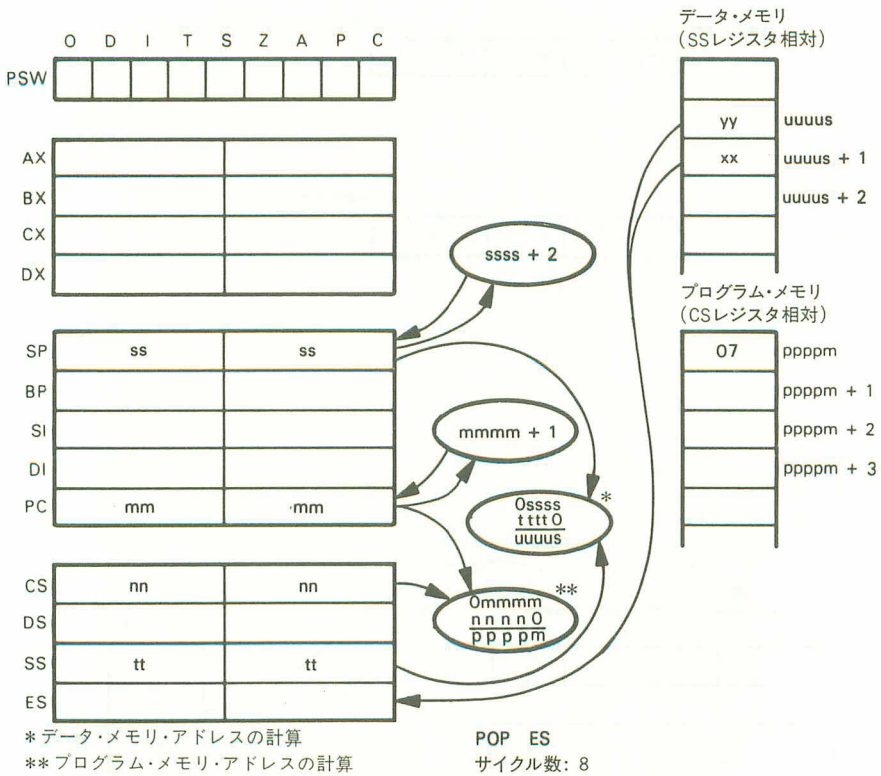
命令コードを次に示す．

たとえば，

POP　ES

の命令は，スタックのトップの２バイトをＥＳレジスタにポップする．rr=01のとき，動作は定義されない．

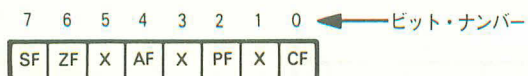
＊データ・メモリ・アドレスの計算
＊＊プログラム・メモリ・アドレスの計算

POP　ES
サイクル数：8

注）

1．この命令は，セグメント・レジスタにのみデータをポップする．8086の他のレジスタにデータをポップするためには，命令　POP reg を参照．
2．POPによって行なわれる機能のより完全な記述については，POP reg を参照．
3．CSに対するPOPは正しくない．CSの変更は，PCへのロードも行なう命令，JMP，CALL，RET，IRET，INTによってのみ行なわれる必要がある．
4．インタラプトは，この命令の終わりではサンプルされない．これに続く命令の終わりにサンプルされる．この制限は，インタラプトを起こさずに，32ビットのポインタ全体の回復を可能にする．これは，SSとSPをロードする際に問題となる．

POPF　（Pop Flags）

スタックのトップからフラグ・レジスタにリードする．

スタックのトップの2バイトを，フラグ・レジスタにポップする．フラグ・レジスタの下位バイトを以下に示す．

2 番目にポップされるバイトは，フラグ・レジスタの上位バイトにストアされる．このバイトの形式を以下に示す．

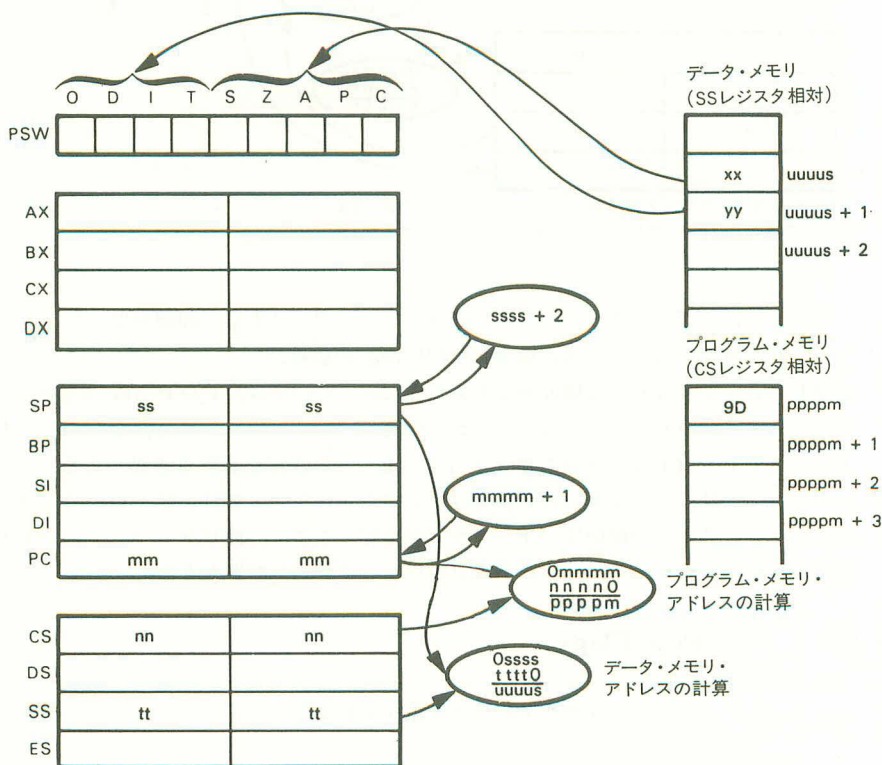
命令コードを次に示す．

$$\underbrace{\text{POPF}}_{\text{9D}}$$

たとえば，スタックのトップの 2 バイトが $4F_{16}$（最上位）と 32_{16} である場合を考える．

POPF

の実行によって，キャリー，パリティ，ゼロ，インタラプトのフラグは 1 になり，その他のフラグは 0 になる．

POPF
サイクル数:8

注）

1．すべてのスタックの操作と同じく，スタック・ポインタが初期設定されていることは重要である．さらに，ＰＯＰＦ命令が実行される前に，フラグの値をストアするためにＰＵＳＨＦ命令が実行されていることが適切である．
2．この命令は，8080の命令　ＰＯＰ　ＰＳＷ　の機能の一部を果たす*.

PUSH　mem/reg　　　(Push)

スタックのトップにライトする．

この命令は，指定されたレジスタあるいはメモリ位置の内容をスタックのトップにプッシュする．これは16ビットのプッシュ操作である．

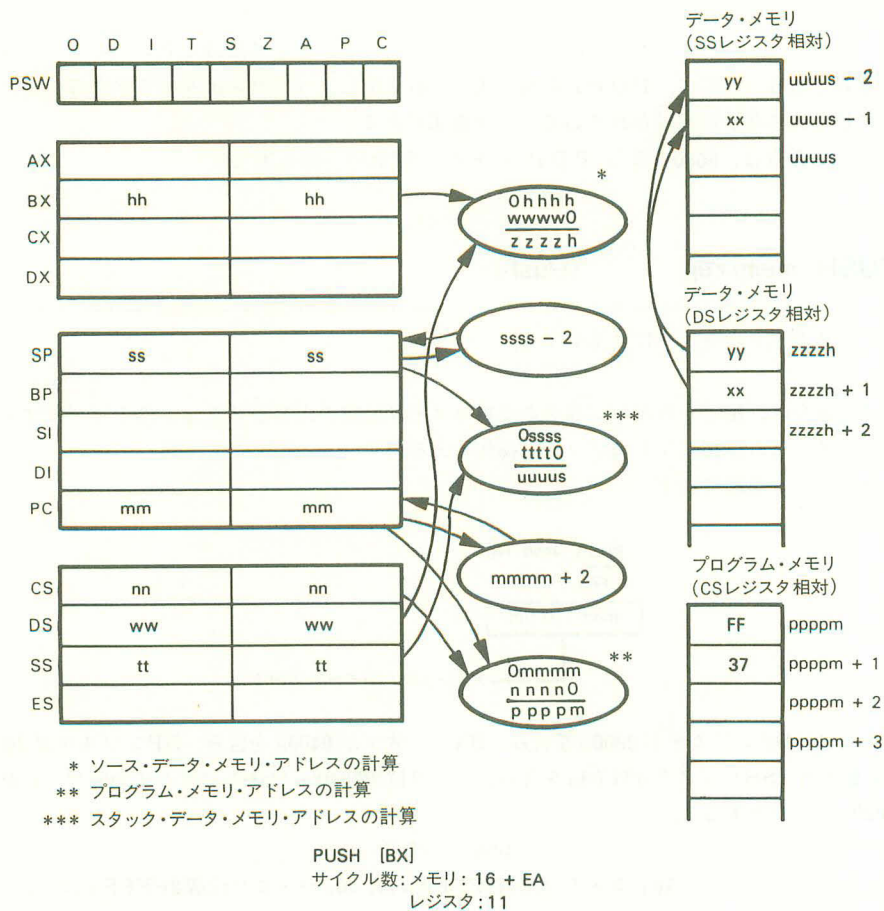
命令コードを次に示す．

たとえば，DSレジスタが 2800_{16} を含み，BXレジスタが 0400_{16} を含み，SPレジスタが 1000_{16} を含み，SSレジスタが $2F00_{16}$ を含み，メモリ位置 28400_{16} にストアされているワードが $A020_{16}$ を含むとすると，

PUSH　〔BX〕

の命令実行によって，$A0_{16}$ がメモリ位置 $2FFFF_{16}$ に，20_{16} がメモリ位置 $2FFFE_{16}$ にストアされる．ＳＰレジスタは $0FFE_{16}$ に調整される．

＊　8080の命令　ＰＯＰ　ＰＳＷでは，8ビットのフラグ・レジスタとアキュムレータへのポップを行なう（訳者注）．

PUSH [BX]
サイクル数:メモリ:16 + EA
レジスタ:11

注)

1．この命令は，一般にレジスタをスタックにプッシュするためには用いられない．命令 PUSH reg はこの機能を果たし，しかもプログラム・メモリの1バイトのみを占有する．
2．ステータスは影響を受けない．

PUSH reg (Push)

スタックのトップにライトする．

この命令は，指定された16ビット・レジスタの内容を，スタックのトップにプッシュする．

命令コードを次に示す．

例として，

PUSH　　SI

の命令を考える．この命令は，ＳＩレジスタの16ビットの内容をスタックにプッシュする．この機能は以下のように行なわれる．

1．スタック・ポインタを1だけ減少．

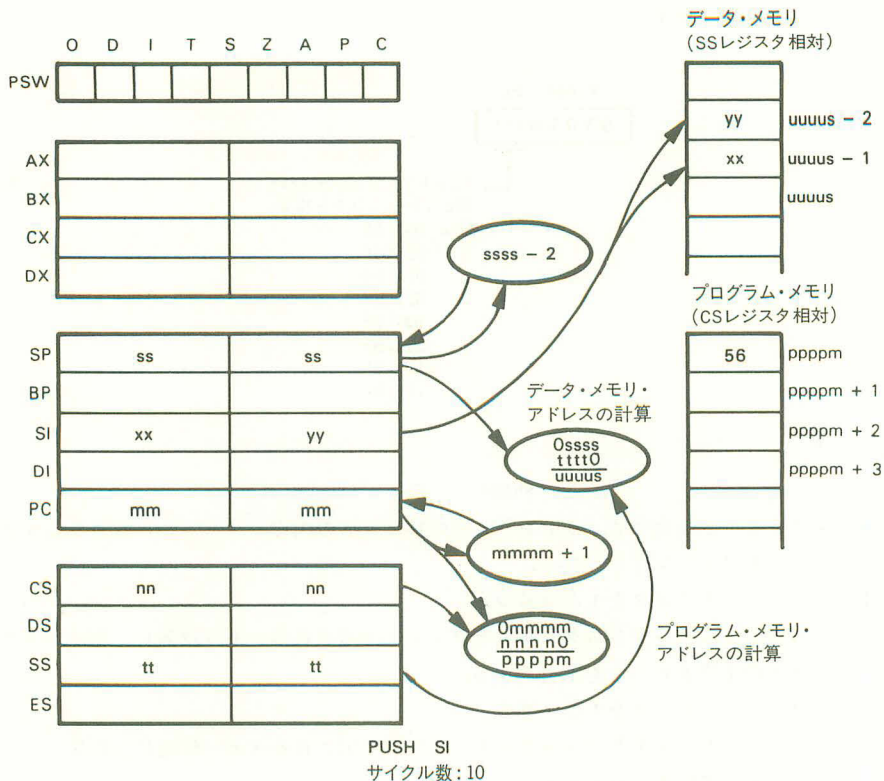
2．スタック・ポインタとスタック・セグメントで示されるメモリ位置に，指定されたレジスタの上位8ビットをストアする．

3．スタック・ポインタを1だけ減少．

4．スタック・ポインタとスタック・セグメントで示されるメモリ位置に，指定されたレジスタの下位8ビットをストアする．

スタック・ポインタは，一般にスタックのトップと呼ばれるスタックにストアされた最後の要素を示している．

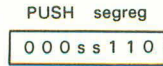
PUSH SI
サイクル数:10

注)

1. この命令は，セグメント・レジスタあるいはフラグ・レジスタのプッシュには使用できない．セグメント・レジスタのプッシュには，命令 PUSH segreg を参照．フラグ・レジスタのプッシュには，PUSHF 命令を参照．
2. この命令は，スタック・ポインタが初期設定された後での使用が最も有効である．事実，この命令は，スタック・ポインタの初期設定後のみ用いられる必要がある．
3. スタックからデータを得るには，POP命令を用いる．
4. この命令は，8080の命令 PUSH reg と同じ機能を果たす．

PUSH segreg (Push)

スタックのトップにライトする．

この命令は，指定された16ビットのセグメント・レジスタの内容を，スタックのトップにプッシュする．

命令コードを次に示す．

たとえば，次の命令を考える．

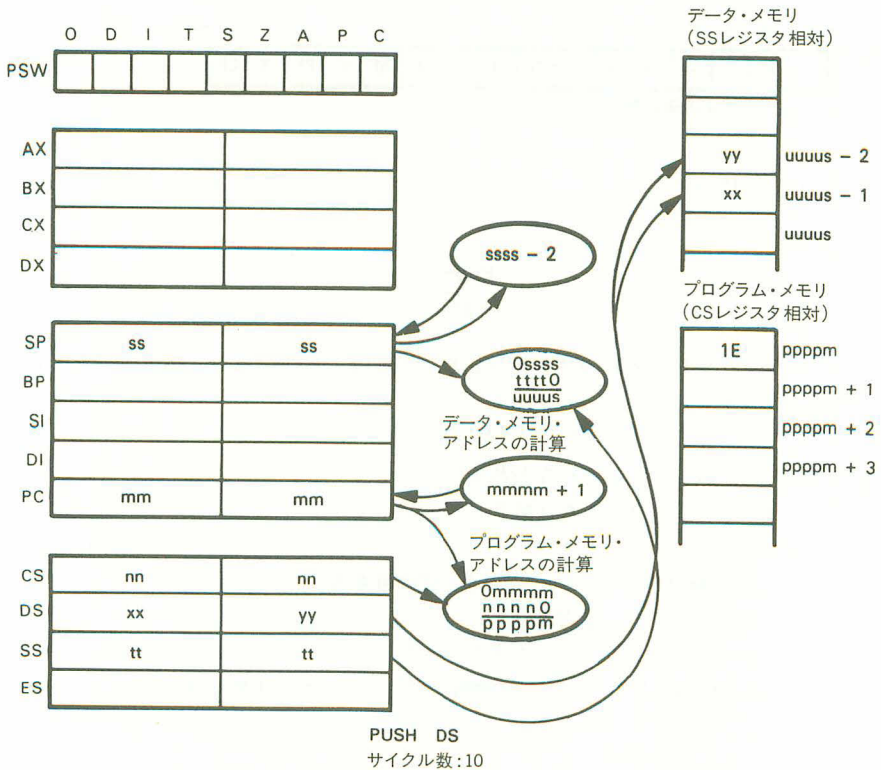
PUSH　DS

この命令は，DSレジスタの16ビットの内容をスタックにプッシュする．

ss＝01のとき，操作は正しくない．

O D I T S Z A P C
PSW
AX
BX
CX
DX
SP　ss　ss
BP
SI
DI
PC　mm　mm
CS　nn　nn
DS　xx　yy
SS　tt　tt
ES
ssss − 2
0ssss
tttt0
uuuus
データ・メモリ・アドレスの計算
mmmm + 1
プログラム・メモリ・アドレスの計算
0mmmm
nnnn0
ppppm
データ・メモリ（SSレジスタ相対）
yy　uuuus − 2
xx　uuuus − 1
uuuus
プログラム・メモリ（CSレジスタ相対）
1E　ppppm
ppppm + 1
ppppm + 2
ppppm + 3

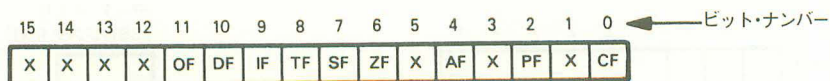
PUSH　DS
サイクル数:10

注）

1．この命令は，セグメント・レジスタの内容をスタックにプッシュするためにだけ用いることができる．他のレジスタの内容をプッシュするためには，ＰＵＳＨ regとＰＵＳＨＦ の命令を参照．

3．適切な結果を保証するためには，スタック・ポインタが初期設定されていなければならないことに注意．

PUSHF　(Push Flags)

フラグ・レジスタの内容をスタックのトップにライトする．

この命令は，フラグ・レジスタの内容をスタックのトップにプッシュする．フラグ・レジスタの形式を以下に示す．

ここでＸは不定の値を表わす．

最初にビット15-8がスタックに，続いてビット7-0がストアされる．

命令コードを次に示す．

PUSHF
9C

例として，ＩＦ，ＳＦ，ＺＦのフラグが1で，一方ＯＦ，ＤＦ，ＴＦ，ＡＦ，ＰＦ，ＣＦのフラグが0とすると，

PUSHF

の実行によって，以下の操作が行なわれる．

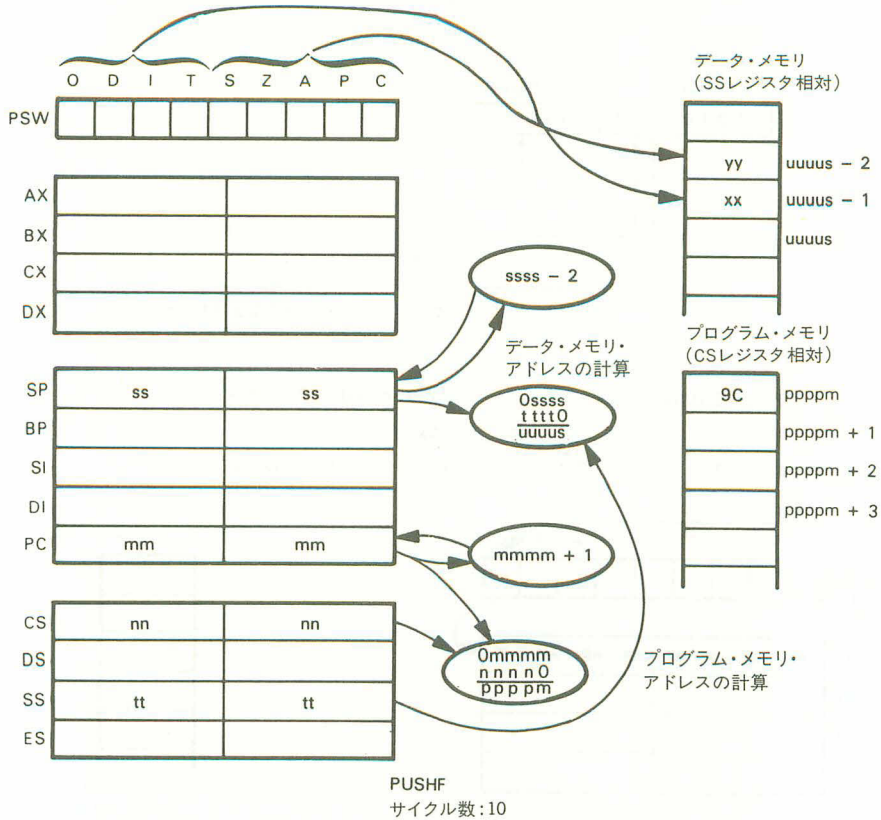
1．スタック・ポインタを1だけ減少．

2．スタック・ポインタとスタック・セグメント・レジスタで示されるメモリ位置に，バイトXXXX0010をストアする（Ｘは不定の値を表わす）．

3．スタック・ポインタを1だけ減少．

4．スタック・ポインタとスタック・セグメント・レジスタで示されるメモリ位置に，バイト11X0X0X0をストアする．8086に対して，これは1個の16ビット転送を行なっている．

PUSHF
サイクル数：10

注)

1．すべてのスタック命令と同じく，この命令はスタック・ポインタの初期設定が行なわれた後で，最も良くその機能を果たすことに注意．

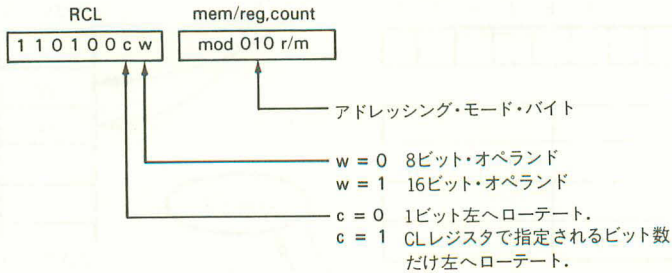
2．この命令は，8080の命令　ＰＵＳＨ　ＰＳＷ　と同じ機能を果たさない．
ＰＵＳＨ　ＰＳＷ　命令は，8080のフラグと同様にアキュムレータの内容をプッシュする．
ＰＵＳＨ　ＰＳＷ　をエミュレイトするには，ＬＡＨＦ　命令を参照．

RCL mem/reg,count　　　　(Rotate through Carry Left)

キャリーと共にレジスタあるいはメモリの内容を左へローテートする．

指定されたビット数だけ，指定されたレジスタあるいはメモリの内容を，キャリーと共に左へローテートする．変数countで表わされるローテートのビット数は，１あるいはＣＬ

レジスタに含まれる数である．８あるいは16ビットのオペランドが指定できる．

命令コードを次に示す．

ＡＸレジスタが FB00₁₆ を含み，キャリー・フラグが０であるとする．

RCL　　AX, 1

の実行後，キャリー・フラグは１になり，ＡＸレジスタは F600₁₆になる．

O D I T S Z A P C

PSW X

AX BX CX DX

SP BP SI DI PC mm mm

CS nn nn DS SS ES

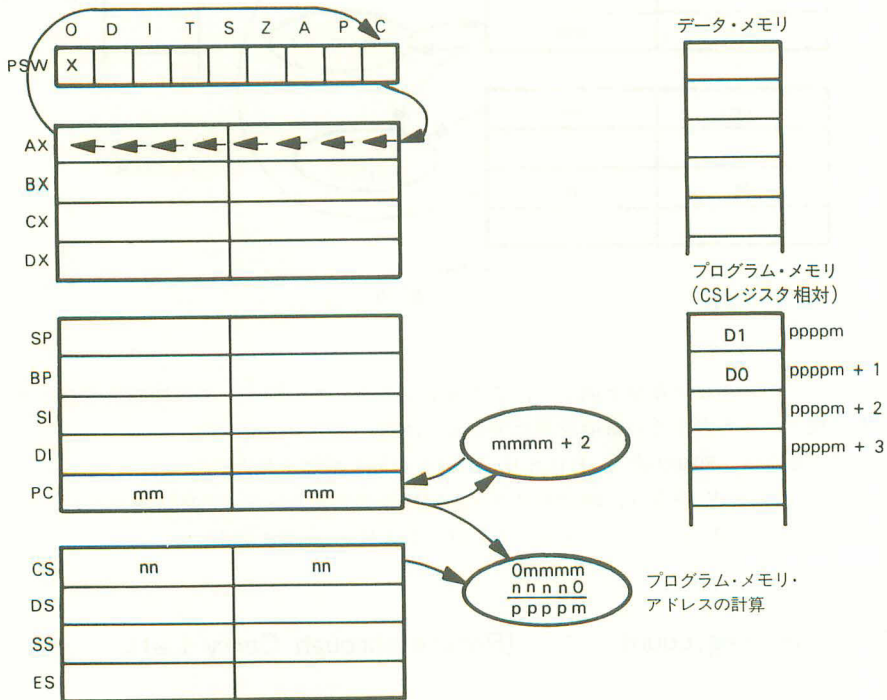
mmmm + 2

0mmmm
nnnn0
ppppm

プログラム・メモリ・
アドレスの計算

データ・メモリ

プログラム・メモリ
(CSレジスタ相対)

D1 ppppm
D0 ppppm + 1
ppppm + 2
ppppm + 3

RCL　AX, 1

サイクル数：レジスタ（1ビットのローテート）：2
レジスタ（Nビットのローテート）：8 + (4 * N)
メモリ（1ビットのローテート）：15 + EA
メモリ（Nビットのローテート）：20 + EA + (4 * N)

注)

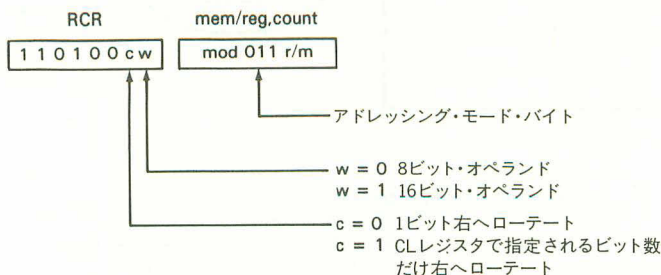
1．この命令は，8080の命令 RAL と同じ機能を果たす．しかしこの命令は，複数ビットのローテートが可能，16ビット数値のローテートが可能，そして任意のレジスタあるいはメモリ位置のローテートができるという点で，非常に大きい融通性がある．

2．8あるいは16ビットのローテートのどちらが実行されるかは，直観的に明らかではないことに注意．これが決定される方法は，用いるアセンブラに依存している．この興味ある問題の解説は，この章の終わりを参照のこと．

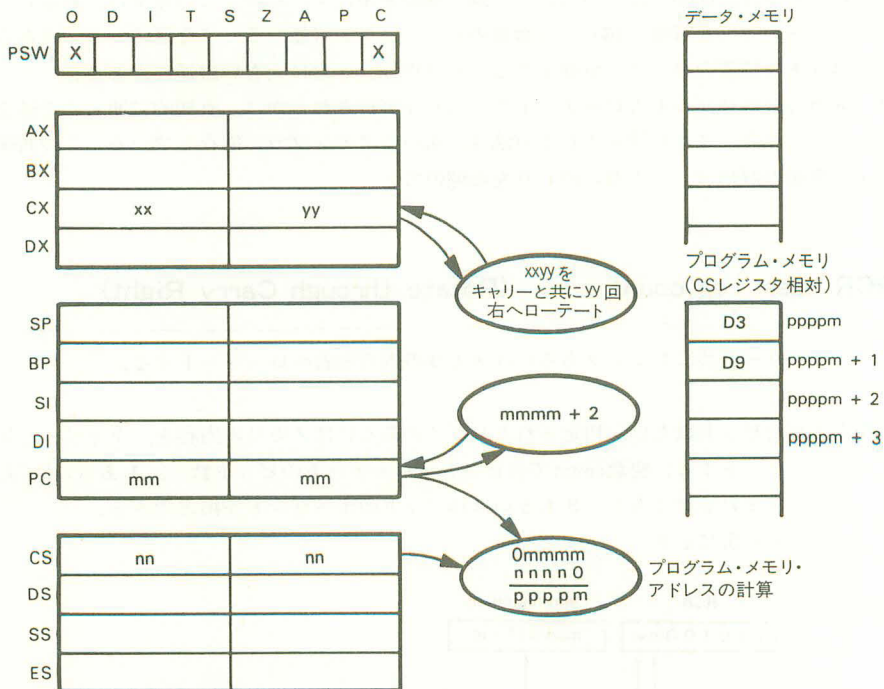
RCR mem/reg,count　　(Rotate through Carry Right)

キャリーと共にレジスタあるいはメモリの内容を右へローテートする．

指定されたビット数だけ，指定されたレジスタあるいはメモリの内容を，キャリーと共に右へローテートする．変数countで表わされるローテートのビット数は，1あるいはCLレジスタに含まれる数である．8あるいは16ビットのオペランドが指定できる．

命令コードを次に示す．

CXレジスタが$F709_{16}$を含み，キャリー・フラグが1であるとする．

RCR　　CX,CL

の実行後，CXレジスタは$09FB_{16}$になり，キャリー・フラグは1になる．

RCR CX,CL

サイクル数：レジスタ（Nビットのローテート）：8 + (4 * N)
レジスタ（1ビットのローテート）：2
メモリ（Nビットのローテート）：20 +EA + (4 * N)
メモリ（1ビットのローテート）：15 + EA

注）

1．この命令は，8080の命令 RAR と同じ機能を果たす．しかしこの命令は，複数ビットのローテートが可能，16ビット数値のローテートが可能，そして任意のレジスタあるいはメモリ位置のローテートができるという点で，非常に大きい融通性がある．

2．この命令を考えると，8あるいは16ビットのローテートの間の差違は明白ではない．この問題の解説は，この章の終わりを参照．

REP/REPE/REPNE/REPNZ/REPZ　　(Repeat)

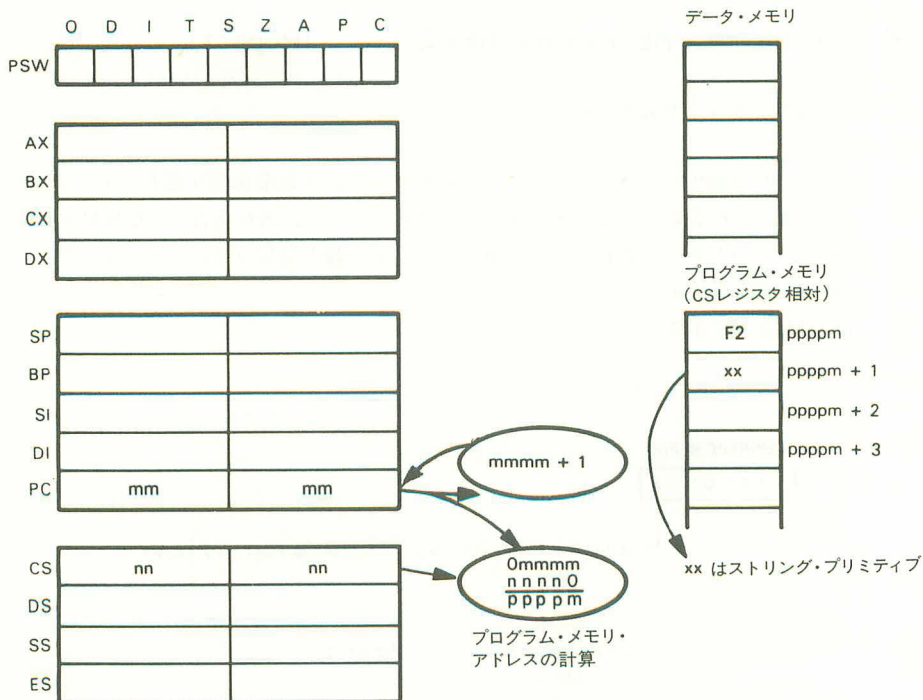
後続のストリング命令を繰り返す．

ＣＸレジスタが減少して０になるまで，後続のストリング命令を繰り返す．すべてのストリング命令は，ＣＸが０になるまで実行を続ける．ただし，ＳＣＡＳ と ＣＭＰＳ の命令は例外で，このときは，ＺＦフラグの値がこの命令の最下位ビット，ｚビットに等しくなくなれば，実行を中止する．

命令コードを次に示す．

以下の一連の命令において，

```
MOV     SI,IOBUF
LES     DI,ADDR
MOV     CX,COUNT
REP
MOVB
```

ＲＥＰ ＭＯＶＢ の命令で，ＣＯＵＮＴで表わされるバイト数がＩＯＢＵＦからＡＤＤＲに移動される．

REP

サイクル数：2：REPプレフィックスだけの処理．
これは後続のストリング・プリミティブ
の繰り返しには含まれない．

注)

1．REPE と REPZ の命令コードは $F3_{16}$ で，REPNE と REPNZ の命令コードは $F2_{16}$ になる．
2．REP は命令プレフィックスと呼ばれる．他のプレフィックスには，LOCK と SEG がある．REP を LOCK あるいは SEG のプレフィックスと共に用いるときは，注意が必要となる．この注意についての詳細は，次章を参照のこと．

RET (Return)

サブルーチン（セグメント間）から復帰する．

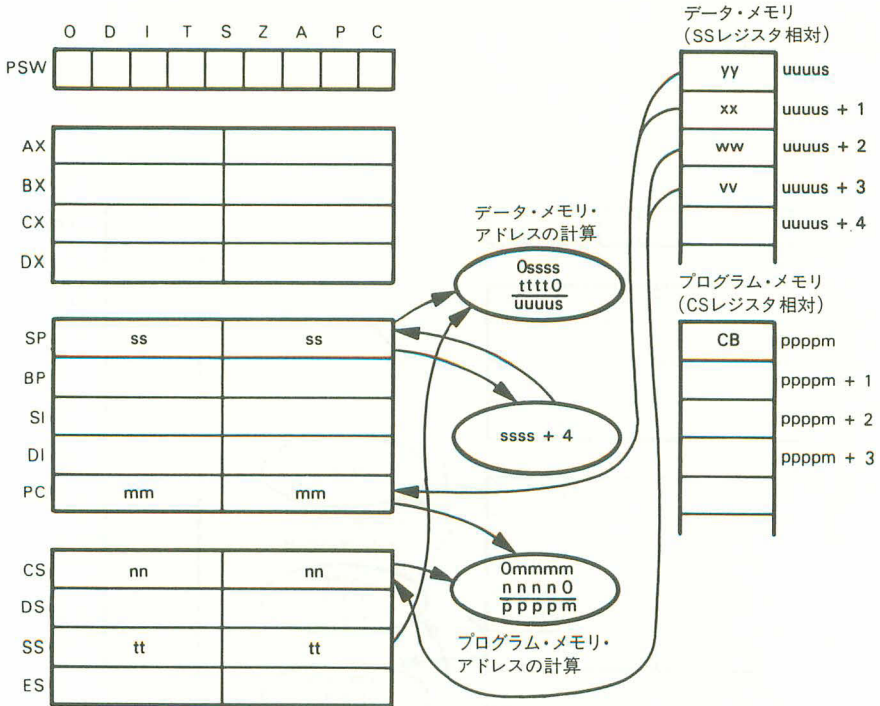
スタックのトップの2バイトを，プログラム・カウンタにポップする．この2バイトは，実行されるべき次の命令のオフセット・アドレスを与える．スタックの次の2バイトをC

Sレジスタにポップする．この2バイトは，実行されるべき次の命令のコード・セグメント・アドレスを与える．以前のプログラム・カウンタとコード・セグメント・レジスタの内容は失なわれる．

命令コードを次に示す．

RET
CB

RET
サイクル数:24

注)

1．すべてのサブルーチンには，少なくとも1個の RET 命令が必要である．この命令は，サブルーチン内で実行される最後の命令であり，制御を呼び出し元プログラムに戻す．

2．この RET 命令は，2つのセグメント間CALL，セグメント間ダイレクトとセグメント間インダイレクトに対応している．

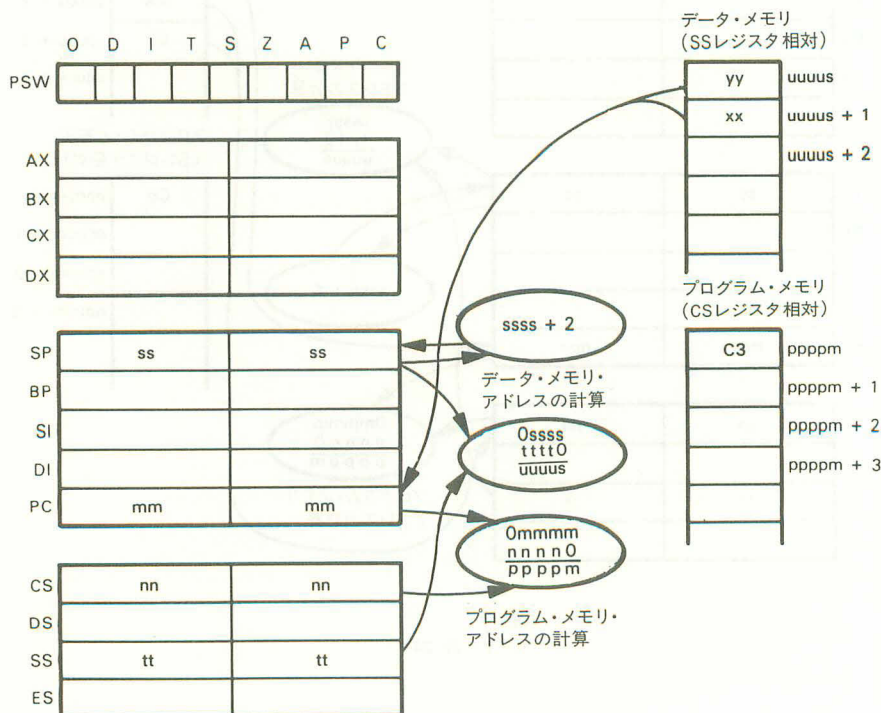
3．ステータスは影響を受けない．

RET　　　　(Return)

サブルーチン（セグメント内）から復帰する．

スタックのトップの2バイトの内容をプログラム・カウンタに移動する．すなわち，スタックをプログラム・カウンタにポップする．このバイトは，実行されるべき次の命令のオフセット・アドレスを与える．以前のプログラム・カウンタの内容は失なわれる．

命令コードを次に示す．

RET
C3

RET
サイクル数：16

注）

1．この命令は，8080の命令 RET と同じ機能を果たす．

2．すべてのサブルーチンには，少なくとも1個の RET 命令が必要である．これは，サブルーチン内で実行される最後の命令で，実行を呼び出し元プログラムに戻す．呼び

出し元へ戻るために他の方法も可能であるが，一般に簡単な RET 命令と比較して能率が悪く不明瞭となる．

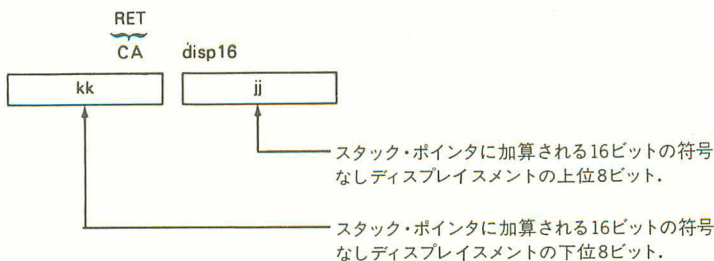
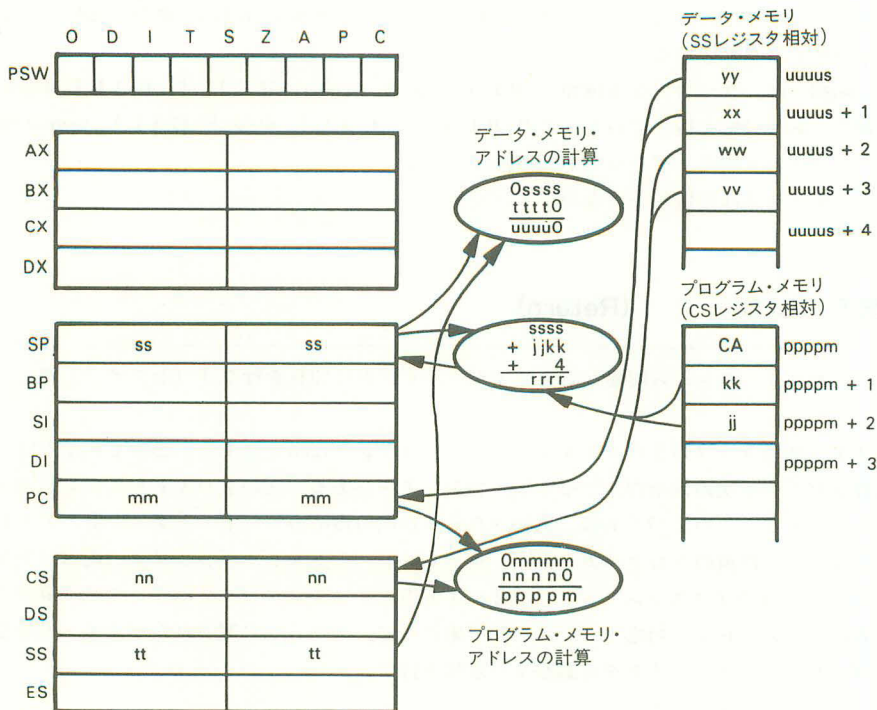
3．8086には，3つの異なる種類の RET がある．これら RET は，CALL命令に対してある対応を持っている．この RET は，CALL disp と CALL mem/reg のセグメント内インダイレクトに対応している．

4．ステータスは影響を受けない．

RET disp16 (Return)

サブルーチンから復帰し，スタック・ポインタに加算を行なう（セグメント間）．

スタックのトップの2バイトをプログラム・カウンタにポップする．この2バイトは，実行されるべき次の命令のオフセット・アドレスを与える．次の2バイトをCSレジスタにポップする．この2バイトは，実行されるべき次の命令のコード・セグメント・アドレスを与える．以前のプログラム・カウンタとコード・セグメント・レジスタの内容は失なわれる．後続のプログラム・メモリの2バイトのデータをスタック・ポインタに加算する．これは，このRETに対応するCALLに先立って，スタックに置かれたパラメータを受け渡したスタック・ポインタを調整する意味を持つ．

命令コードを次に示す．

RET jjkk
サイクル数:23

注)

1．ステータスは影響を受けない．

2．すべてのサブルーチンには，少なくとも1つのRET命令が必要である．この命令は，サブルーチン内で実行される最後の命令で，呼び出し元プログラムにおいて対応するCALLの次の命令から実行を再開する．

3．このRET命令は2つのセグメント間CALL，セグメント間ダイレクトとセグメント間インダイレクトに対応する．

RET disp16 (Return)

サブルーチンから復帰し，スタック・ポインタに加算を行なう（セグメント内)．

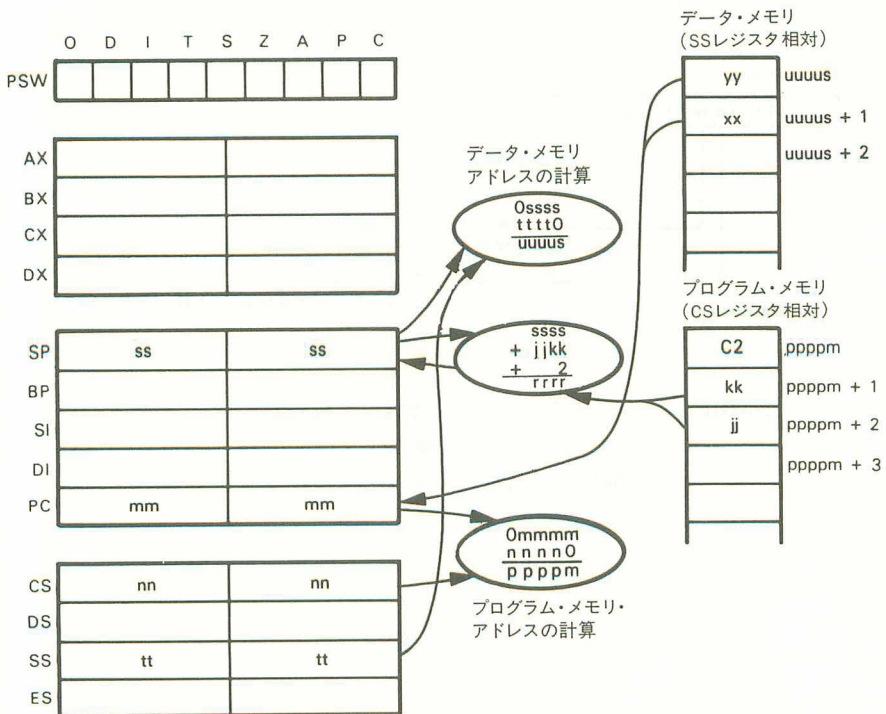
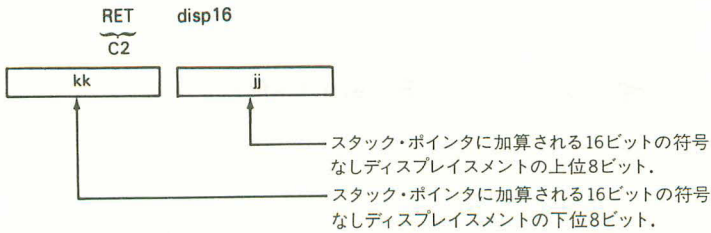
スタックからプログラム・カウンタにポップする．移動する2バイトは，実行されるべき次の命令のオフセット・アドレスを与える．以前のプログラム・カウンタの内容は失なわれる．後続のプログラム・メモリの2バイトのデータをスタック・ポインタに加算する．

これは，このＲＥＴに対応するＣＡＬＬに先立って，スタックに置かれたパラメータを受け渡したスタック・ポインタを調整する意味を持つ．

命令コードを次に示す．

データ・メモリ（SSレジスタ相対）
O D I T S Z A P C
PSW
yy uuuus
xx uuuus + 1
uuuus + 2
AX
BX
CX
DX
データ・メモリ・アドレスの計算
0ssss
tttt0
uuuus
プログラム・メモリ（CSレジスタ相対）
SP ss ss
ssss
+ jjkk
+ 2
rrrr
C2 ppppm
BP
kk ppppm + 1
SI
jj ppppm + 2
DI
ppppm + 3
PC mm mm
0mmmm
nnnn0
ppppm
CS nn nn
DS
プログラム・メモリ・アドレスの計算
SS tt tt
ES

RET　jjkk
サイクル数：20

注）

1．すべてのサブルーチンには，少なくとも1個のＲＥＴ命令が必要である．これは，サ

ブルーチン内で実行される最後の命令であり，呼び出し元プログラムに実行を戻す．

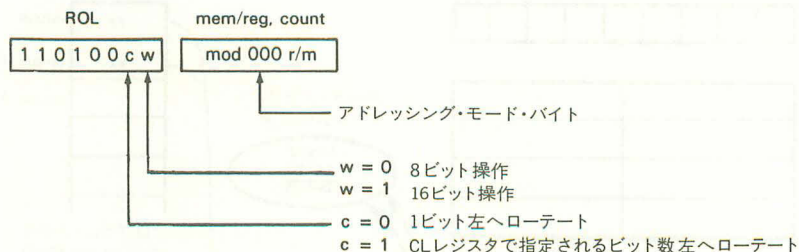
2．8086には，3つの異なるＲＥＴＵＲＮ命令がある．これらのＲＥＴＵＲＮにはＣＡＬＬ命令との対応がある．このＲＥＴは，ＣＡＬＬ disp とＣＡＬＬ mem/regのセグメント内インダイレクトに対応している．

3．ステータスは影響を受けない．

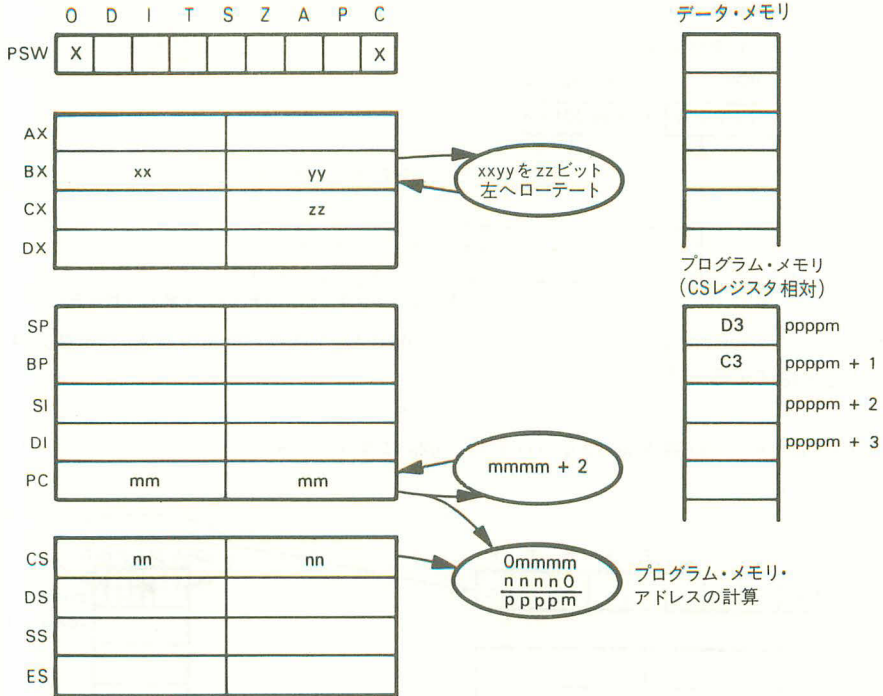
ROL mem/reg, count (Rotate Left)

レジスタあるいはメモリの内容を左へローテートする．

指定されたビット数だけ，指定されたレジスタあるいはメモリ位置の内容を左へローテートする．変数 count で表わされるローテートのビット数は，1あるいはＣＬレジスタに含まれる数である．

命令コードを次に示す．

BXレジスタが$AB1F_{16}$を含み，ＣＬレジスタが03_{16}を含むとする．

ROL　BX, CL

の実行後，ＢＸレジスタは$58FD_{16}$になり，キャリー・フラグは1になる．

ROL　BX.CL

サイクル数：レジスタ(Nビットのローテート)：	8 + (4 * N)
レジスタ(1ビットのローテート)：	2
メ　モ　リ(Nビットのローテート)：	20 + EA + (4 * N)
メ　モ　リ(1ビットのローテート)：	15 + EA

注)

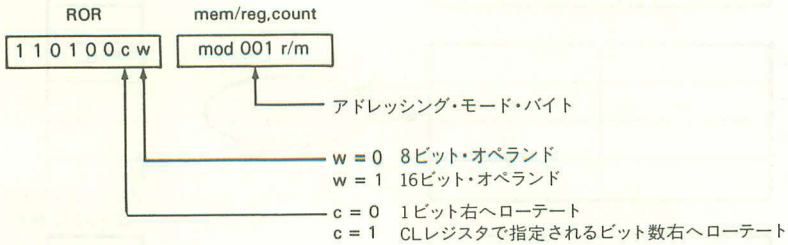
1．この命令は，8080の命令 RLC と同じ機能を果たす．しかしこの命令は，複数ビットのローテートが可能，16ビット数値のローテートが可能，そして任意のレジスタあるいはメモリ位置のローテートができるという点で，非常に大きい融通性がある．
2．この命令の文法を考慮しても，8あるいは16ビットの数値のどちらがローテートされるかは直ちに明らかとはならない．用いるアセンブラは，この困難をどのように解決するかに関して，多くの処理を必要とする．

ROR　mem/reg, Count　　　(Rotate Right)

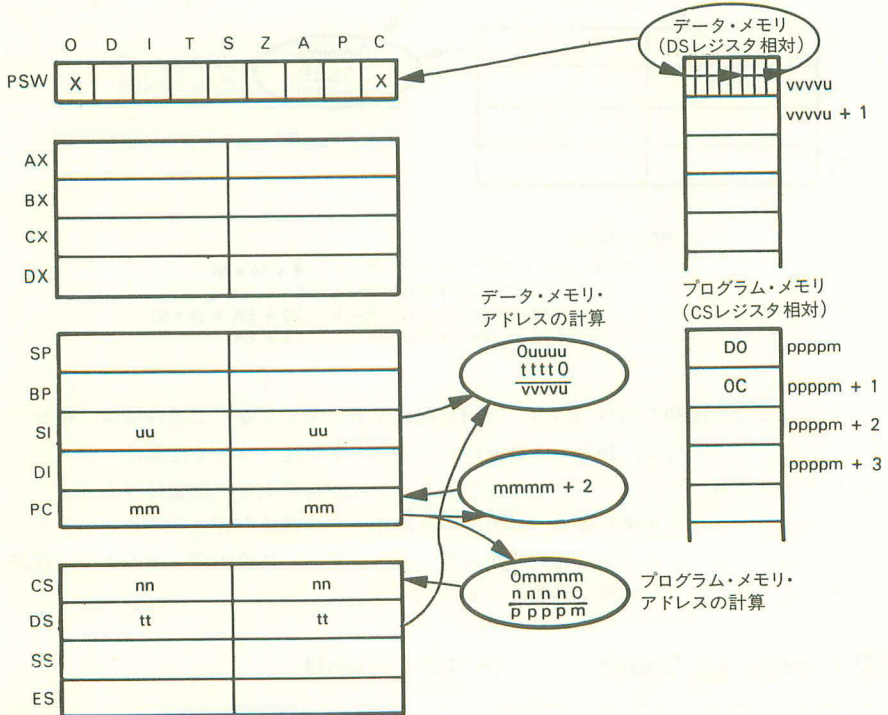
レジスタあるいはメモリの内容を右へローテートする．

指定されたビット数だけ，指定されたレジスタあるいはメモリの内容を右へローテートする．変数 count で表わされるローテートのビット数は，1あるいはCLレジスタに含ま

れる数である．

命令コードを次に示す．

DSレジスタがF000$_{16}$を含み，SIレジスタが06B2$_{16}$を含み，メモリ位置F06B2$_{16}$のバイトが04$_{16}$であるとする．

ROR　〔SI〕, 1

の実行後，メモリ位置F06B2$_{16}$のバイトは02$_{16}$になり，キャリーとオーバーフローのフラグは0となる．

ROR　[SI],1

サイクル数：メ モ リ(1ビットのローテート)：15 + EA
メ モ リ(Nビットのローテート)：20 + EA + (4 * N)
レジスタ(1ビットのローテート)：2
レジスタ(Nビットのローテート)：8 + (4 * N)

注)

1．この命令は，8080の命令 ＲＲＣ と同じ機能を果たす．しかしこの命令は，複数ビットのローテートが可能，16ビット数値のローテートが可能，そして任意のレジスタあるいはメモリ位置のローテートができるという点で，非常に大きい融通性がある．

2．この命令からは，８あるいは16ビット数値のどちらがローテートされるかを決定できないことに注意．

SAHF (Store AH into 8080 Flags)

ＡＨレジスタを8080のフラグにストアする．

この命令は，ＡＨレジスタの内容をフラグ・レジスタの下位８ビットに移動する．ＡＨレジスタのビットは以下のように用いられる．

ビット７：ＳＦフラグにストア
ビット６：ＺＦフラグにストア
ビット５：無視
ビット４：ＡＦフラグにストア
ビット３：無視
ビット２：ＰＦフラグにストア
ビット１：無視
ビット０：ＣＦフラグにストア

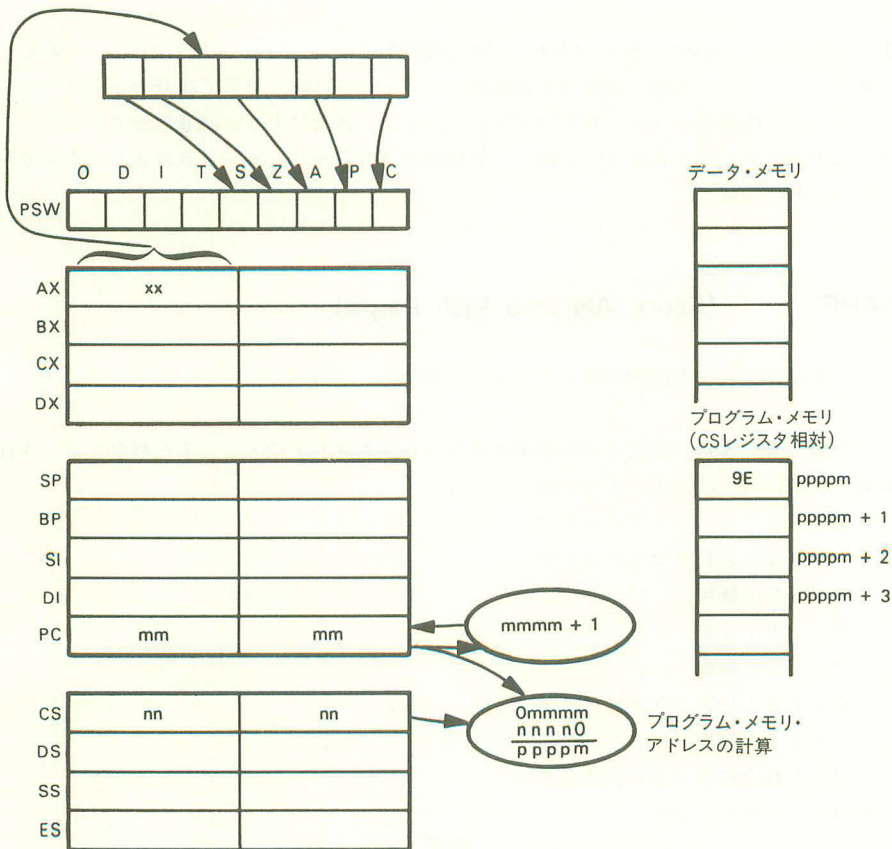
命令コードを次に示す．

SAHF

9E

たとえば，ＡＨレジスタがＥ7_{16}を含むとする．

SAHF

の実行によって，ＳＦ，ＺＦ，ＰＦ，ＣＦのフラグは１になり，一方ＡＦフラグは０になる．

SAHF
サイクル数:4

注)

1．フラグ・レジスタを除いて，レジスタは影響を受けない．OF，DF，IF，TFのフラグは影響を受けない．

2．この命令は，8080の命令 POP PSW をエミュレイトするために，POP AX と共に用いられる．

8086コード	8080コード
POP AX	POP PSW
SAHF	

この8086の命令が意味を持つためには，

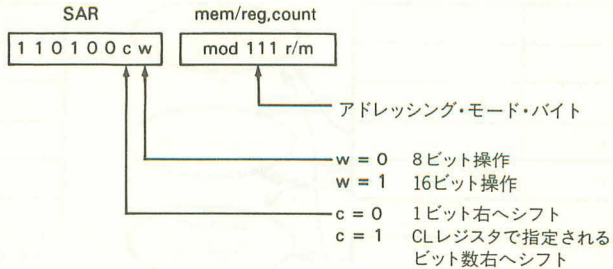
```
LAHF
PUSH AX
```

の命令が8080のフラグをセーブするために用いられる必要があることに注意．

SAR mem/reg, count　　(Shift Arithmetic Right)

レジスタあるいはメモリの内容を右へシフトする.

指定されたビット数だけ, 指定されたレジスタあるいはメモリ位置の内容を右へシフトする. 変数 count で表わされるシフトのビット数は, 1あるいはCLレジスタに含まれる数である. これは算術的右へのシフトである.

命令コードを次に示す.

CLレジスタが 05_{16} を含み, DIレジスタが $180A_{16}$ を含み, DSレジスタが $F800_{16}$ を含み, メモリ位置 $F980A_{16}$ のワードが 0064_{16} であるとする.

SAR 〔DI〕, CL

の実行後, メモリ位置 $F980A_{16}$ のワードは 0003_{16} になる.

SAR [DI],CL

サイクル数:メ モ リ(Nビットのシフト): 20 + EA + (4 * N)
メ モ リ(1ビットのシフト): 15 + EA
レジスタ(Nビットのシフト): 8 + (4 * N)
レジスタ(1ビットのシフト): 2

注)

1. 論理的右へのシフトに対して,これは算術的右へのシフトである.その違いを以下に示す.

算術的右へのシフト(SAR)

すべてのビットを右へ一度にシフトする.最上位ビットは同じ状態のままにしておく.これは,最上位ビットの符号拡張の意味を持つ.複数ビットのシフトならば,必要なだけ最上位ビットを符号拡張する.

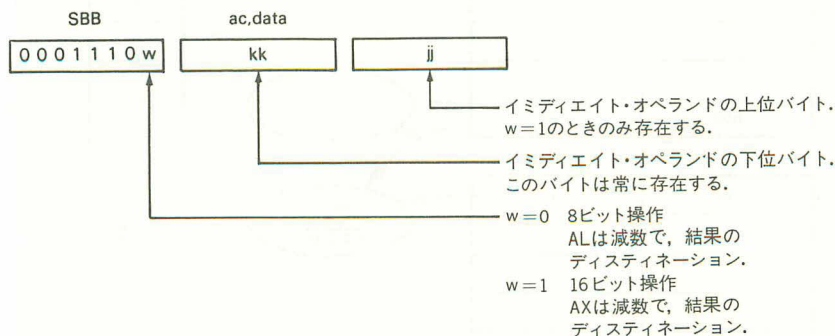
論理的右へのシフト(SHR)

すべてのビットを右へ一度にシフトする.最上位ビットには0をシフトさせる.複数ビットのシフトならば,必要なだけ0をシフトさせることを続ける.

SBB ac, data　　(Subtract with Borrow)

ＡＸあるいはＡＬレジスタからボローと共にイミディエイトを減じる．

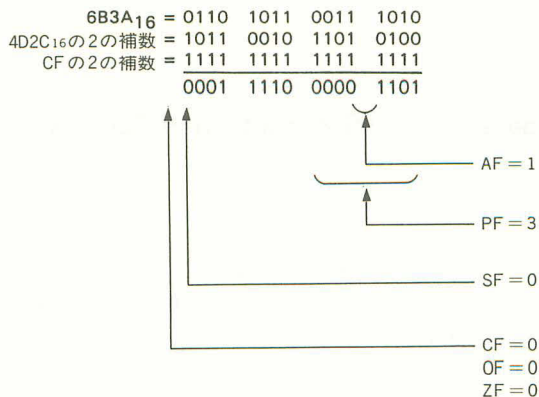
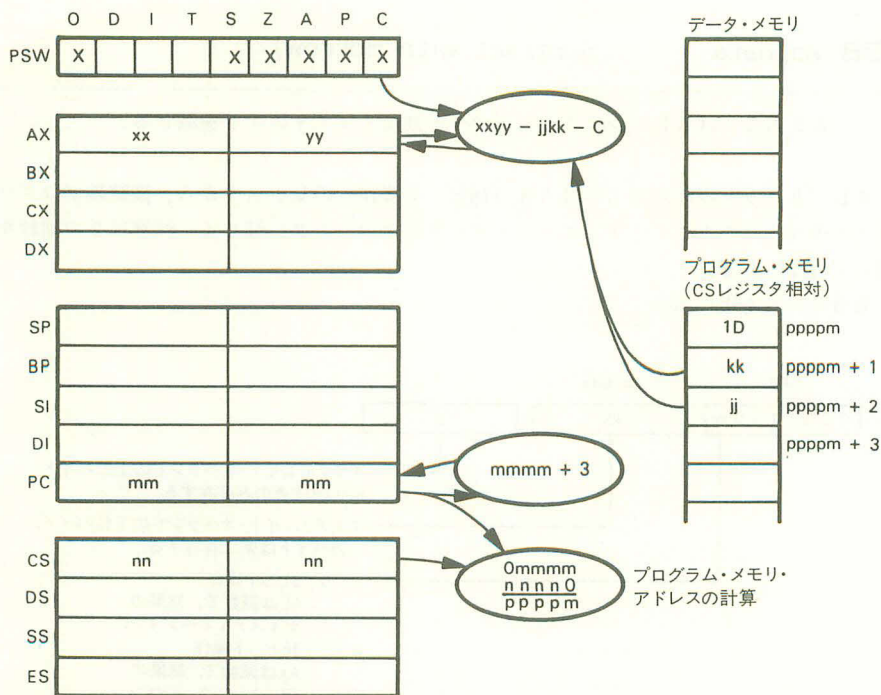
ＡＬ（8ビット操作）あるいはＡＸ（16ビット操作）のレジスタから，後続のプログラム・メモリ・バイトのイミディエイト・データをボローと共に減じる．減算は2の補数を用いて行なわれる．

命令コードを次に示す．

ＡＸレジスタが$6B3A_{16}$を含み，キャリー・フラグが1であるとする．

SBB　AX,4D2CH

の実行後，ＡＸレジスタは$1E0D_{16}$になる．

SBB AX,jjkk
サイクル数:4

注)

1．この命令は，8080の命令 ＳＢＩ data と同じ機能を果たすが，16ビットの操作も可能である．

SBB mem/reg, data (Subtract with Borrow)

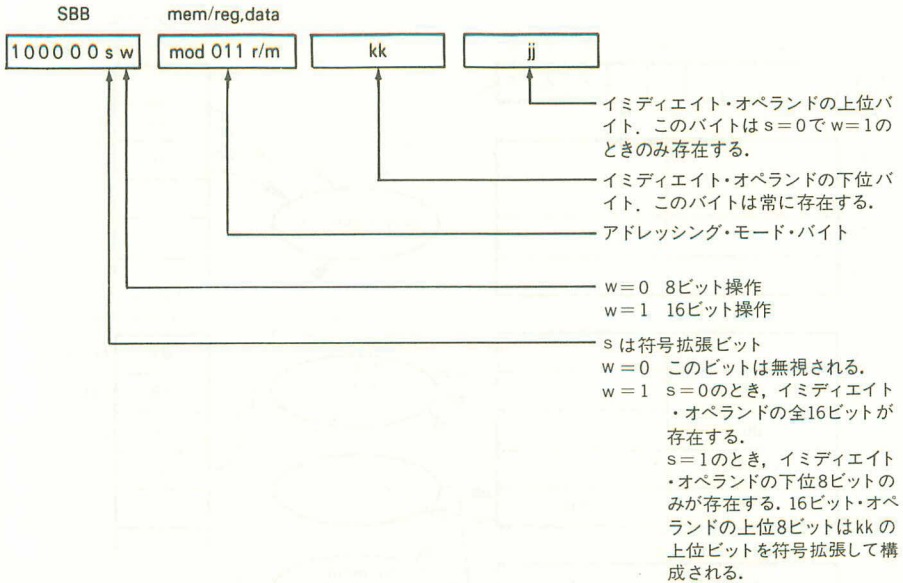
レジスタあるいはメモリからボローと共にイミディエイト・データを減じる．

指定されたレジスタあるいはメモリ位置から，後続のプログラム・メモリ・バイトのイミディエイト・データをボローと共に減じる．8あるいは16ビットの操作が指定できる．減算は，2の補数を用いて行なわれる．

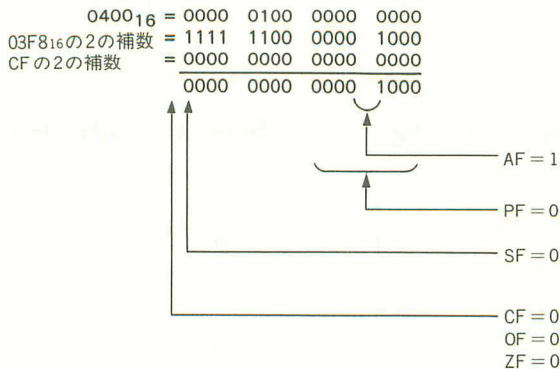
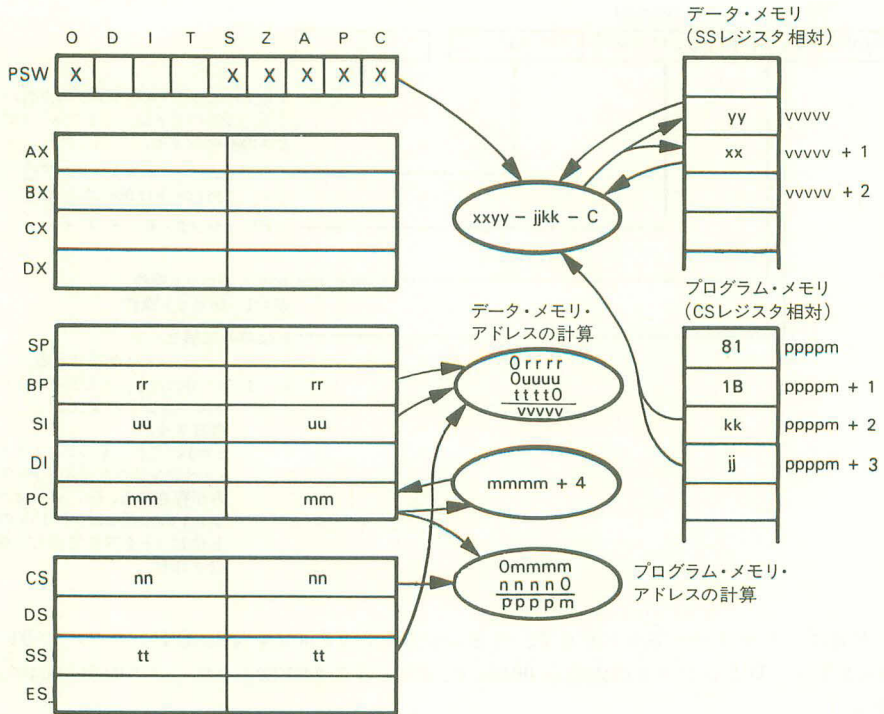
命令コードを次に示す．

SBB　　mem/reg,data

1 0 0 0 0 0 s w	mod 011 r/m	kk	jj

- jj：イミディエイト・オペランドの上位バイト．このバイトはs＝0でw＝1のときのみ存在する．
- kk：イミディエイト・オペランドの下位バイト．このバイトは常に存在する．
- mod 011 r/m：アドレッシング・モード・バイト
- w＝0　8ビット操作
w＝1　16ビット操作
- sは符号拡張ビット
w＝0　このビットは無視される．
w＝1　s＝0のとき，イミディエイト・オペランドの全16ビットが存在する．
s＝1のとき，イミディエイト・オペランドの下位8ビットのみが存在する．16ビット・オペランドの上位8ビットはkkの上位ビットを符号拡張して構成される．

たとえば，キャリー・フラグが0で，SSレジスタが$2F00_{16}$を含み，BPレジスタが$0F6A_{16}$を含み，DIレジスタの内容が0018_{16}で，メモリ位置$2FF82_{16}$のワードの内容が0400_{16}ならば，

SBB　〔BP+SI〕, 03F8H

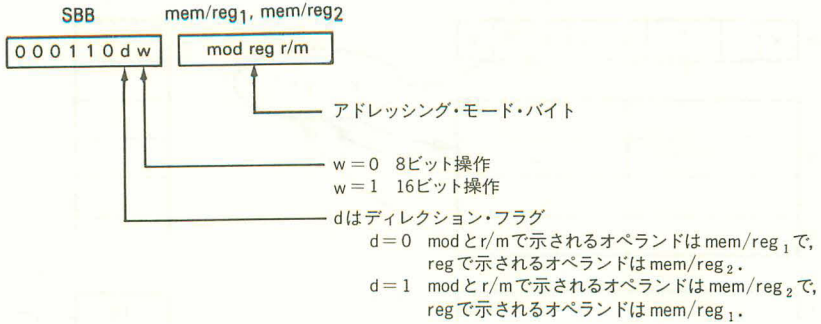
の実行によって，メモリ位置$2FF82_{16}$のワードは0008_{16}に変えられる．

SBB [BP + SI],jjkk

サイクル数:メモリ・オペランド:17+EA
レジスタ・オペランド:4

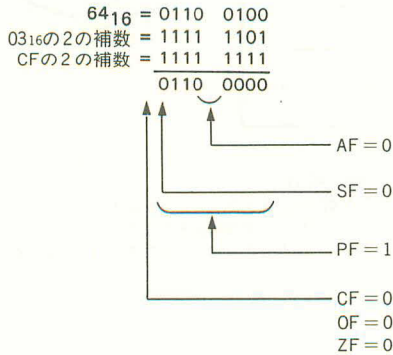
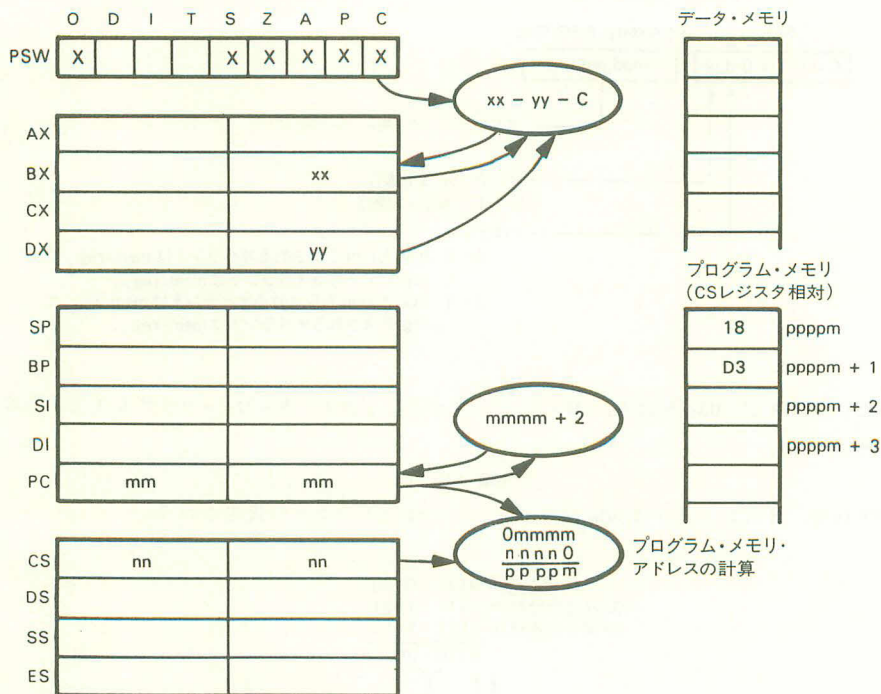
SBB mem/reg$_1$, mem/reg$_2$ (Subtract with Borrow)

レジスタからレジスタを メモリからレジスタを レジスタからメモリを	ボローと共に減じる

mem/reg$_1$ で示されるレジスタあるいはメモリ位置の内容から,mem/reg$_2$で示されるレジスタあるいはメモリ位置の内容とキャリー・フラグを減じる.8あるいは16ビットの操作が指定できる.mem/reg$_1$ あるいはmem/reg$_2$ はメモリ・オペランドであるが,オペランドの一方はレジスタ・オペランドでなければならない.

命令コードを次に示す.

ＤＬレジスタが，03_{16}を含み，ＢＬレジスタが64_{16}を含み，キャリー・フラグが1である場合を考える．

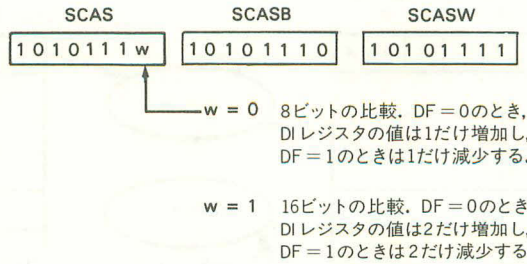
SBB　BL,DL

の実行後，ＢＬレジスタは60_{16}になり，フラグは以下のように設定される。

SBB BL,DL
サイクル数：レジスタからレジスタに対して：3
レジスタからメモリに対して：16 + EA
メモリからレジスタに対して：9 + EA

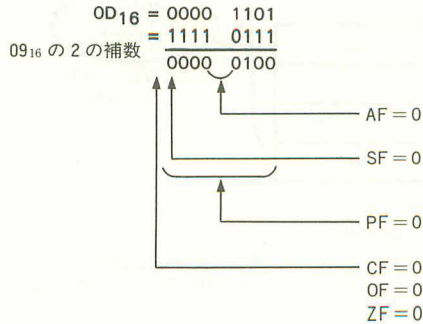
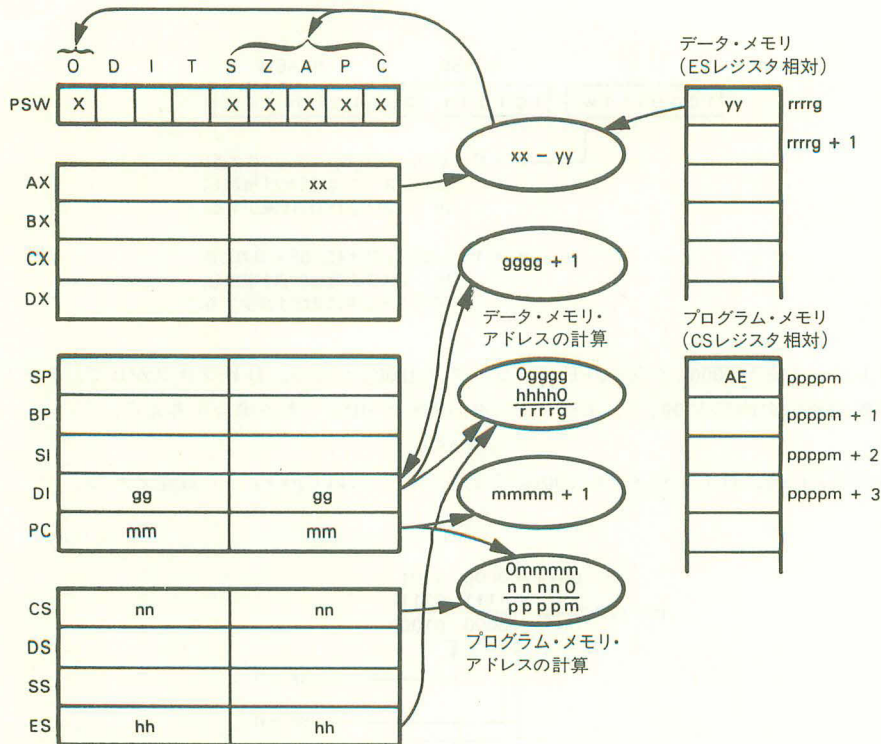
SCAS/SCASB/SCASW (Scan String)

ＡＬあるいはＡＸのレジスタとメモリとを比較する．

ＡＬ（８ビット操作）あるいはＡＸ（16ビット操作）のレジスタと，ＤＩレジスタで示されるメモリ位置の内容とを比較する．比較は，ＡＬあるいはＡＸのレジスタから，ＤＩレジスタで示されるメモリ位置の内容を減じ，その結果をフラグに設定することによって行なわれる．メモリあるいはＡＸレジスタのどちらも変化しない．ＤＩレジスタは，ＤＦフラグの値に応じて増減する．

命令コードを次に示す．

ＤＩレジスタが 0000_{16} を含み，ＥＳレジスタが 1800_{16} を含み，ＤＦフラグが0で，メモリ位置 18000_{16} の内容が 09_{16} で，ＡＬレジスタの内容が $0D_{16}$ である場合を考える．

SCASB

の命令実行後，ＤＩレジスタは 0001_{16} になり，フラグは以下のように設定される．

SCASB

サイクル数:15:1度だけ実行したとき

9＋15＊R:REP プレフィックスによってR回実行したとき

注）

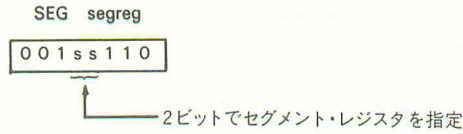
1．ＲＥＰプレフィックスやＬＯＣＫプレフィックスはこの命令と共に用いられる．ＲＥＰプレフィックスとＬＯＣＫプレフィックスがこの命令と共に用いられた場合は問題となる．この問題の解析は次章に示されている．

2．汎用形ＳＣＡＳのバイトあるいはワードのオペランドの決定は，この章の終わりで論じている．

SEG segreg (Segment)

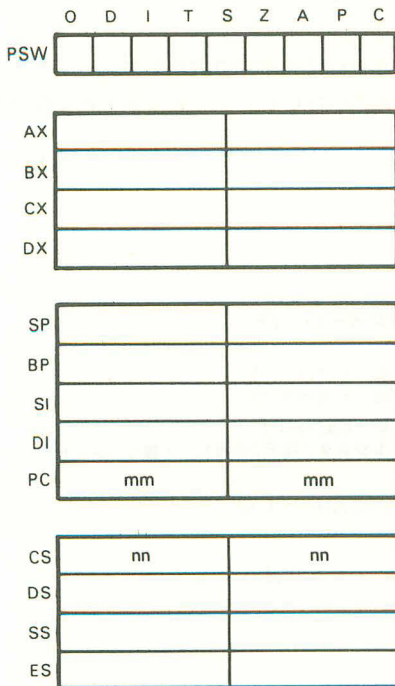
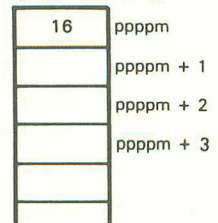
デフォルト・セグメント・レジスタを変更する．

このプレフィックスが先行した命令のデータ・メモリ・アドレスの計算に，指定されたセグメント・レジスタを用いる．すなわち，データ・メモリ・アドレスの計算に，セグメント・アドレスとして指定されたセグメント・レジスタの内容を用いる．

命令コードを次に示す．

DSレジスタが1000_{16}を含み，ESレジスタが2000_{16}を含み，BXレジスタが0008_{16}を含み，メモリ位置10008_{16}のワードが$FEFE_{16}$で，メモリ位置20008_{16}のワードが$060A_{16}$である場合を考える．

```
SEG  ES
MOV  AX,[BX]
```

の実行後，AXレジスタは$060A_{16}$になる．

O D I T S Z A P C
PSW
AX
BX
CX
DX
SP
BP
SI
DI
PC　mm　mm
CS　nn　nn
DS
SS
ES
mmmm + 1
0mmmm
nnnn0
ppppm
プログラム・メモリ・
アドレスの計算
データ・メモリ
プログラム・メモリ
(CSレジスタ相対)
16　ppppm
ppppm + 1
ppppm + 2
ppppm + 3

SEG ES
サイクル数：2

注）

1．セグメント変更プレフィックスの導入はアセンブラに依存する．前に示したように，ＤＳをＥＳに変更するために，

```
SEG ES
MOV AX,〔BX〕
```

が用いられる．

インテルのアセンブラで用いられている他の方法は，

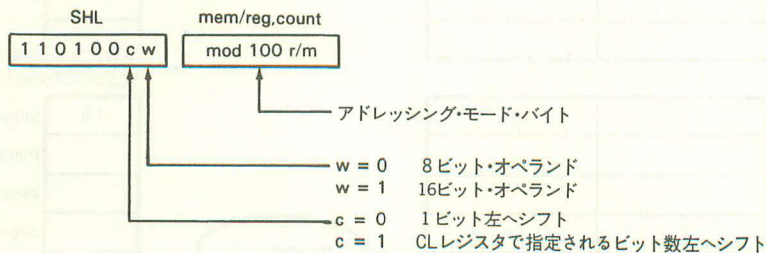
```
MOV AX,ES:〔BX〕
```

で，セグメント変更を移動命令中に組み込むことを要求している．アセンブラは，ＭＯＶ命令コード生成の一部として変更プレフィックスを生成する．

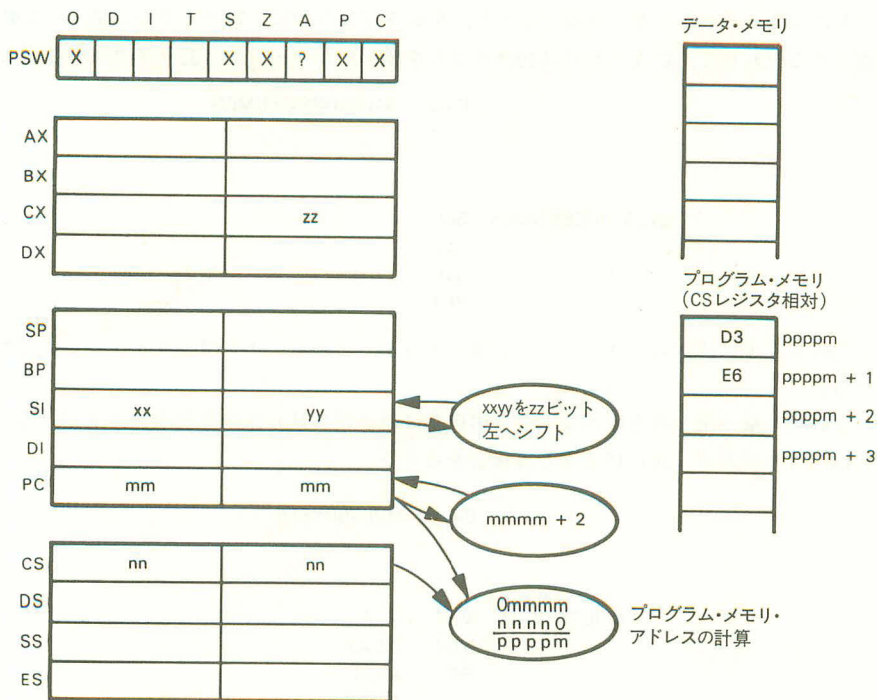
SHL men/reg, count　SAL mem/reg, count　(Shift Left)

レジスタあるいはメモリの内容を左へシフトする．

指定されたビット数だけ，指定されたレジスタあるいはメモリの内容を左へシフトする．変数 count で表わされるシフトのビット数は，１あるいはＣＬレジスタに含まれる数である．これは論理的な左へのシフトである。

命令コードを次に示す．

ＣＬレジスタが02_{16}を含み，ＳＩレジスタが$A450_{16}$を含むとする．

```
SHL SI,CL
```

の実行後，ＳＩレジスタは9140_{16}になり，キャリー・フラグは０になる．

SHL　SI,CL
サイクル数:　レジスタ(Nビットのシフト): 8 + (4 * N)
　　　　　　レジスタ(1ビットのシフト): 2
　　　　　　メモリ　(Nビットのシフト): 20 + EA + (4 * N)
　　　　　　メモリ　(1ビットのシフト): 15 + EA

注)

1. この命令は，シフトによる加算で乗算を行なうために用いることができる．MULとIMULの命令は，実行に少なくとも71サイクルを必要とし，乗算を行なうのにシフトを用いるのが興味ある方法となる場合がある．代表的には，このような場合は，メモリの保存よりも処理速度の最適化が大きい要素となるときや，実行されるべき乗算が常に2のべき乗あるいは常に定数のときに生じる．以下に示すいくつかの場合を考える．

```
                CALL  MULT$BY$8
                 -
                 -
                 -
MULT$BY$8       MOV   CL,3
                SAL   AX,CL
                RET
```

MULTBY 8のルーチンはそのルーチンのコードに5バイトを必要とし，CALLのコードには3バイトを必要とする．しかし，乗算を行なうのに，71サイクル（最小）を必要とする代わりに，CALLには19サイクルを要し，ルーチンには32サイクルを要する。

```
                CALL  SAL$THREE$TIMES
                 -
                 -
                 -
SAL$THREE$TIMES SAL
                SAL
                SAL
                RET
```

上記のルーチンはさらに2バイトを必要とするが，このルーチンは14サイクルで実行される。

2のべき乗を乗じるだけのルーチンの選択は確かにSHL命令を引き立たせていることは明らかである。次に15を乗じる場合を考える．

```
            CALL  MULT$BY$15
             -
             -
MULT$BY$15  MOV   CL,4
            MOV   DX,AX
            SAL   AL,CL
            SUB   AX,DX
            RET
```

このルーチンは，9バイトのコードと41サイクル，それにCALLの19サイクルを必要とする．これは，MUL命令を用いるよりもほんのぎりぎりだけ速い．このルーチンは，個々のSAL命令を含むことによってずっと速く動作できる．

```
            CALL  MULT$BY$15
             -
             -
MULT$BY$15  MOV   DX,AX
            SAL
            SAL
            SAL
            SAL
            SUB   AX,DX
            RET
```

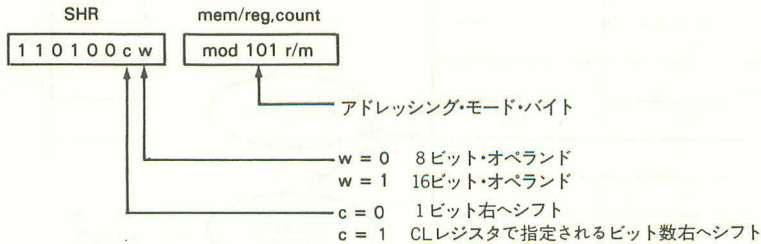
この場合，ルーチンは動作に21サイクルしか要しない．

2．8あるいは16ビットのどちらのローテーションかは，この命令の記述に用いられている表現方法では，指定されていない．

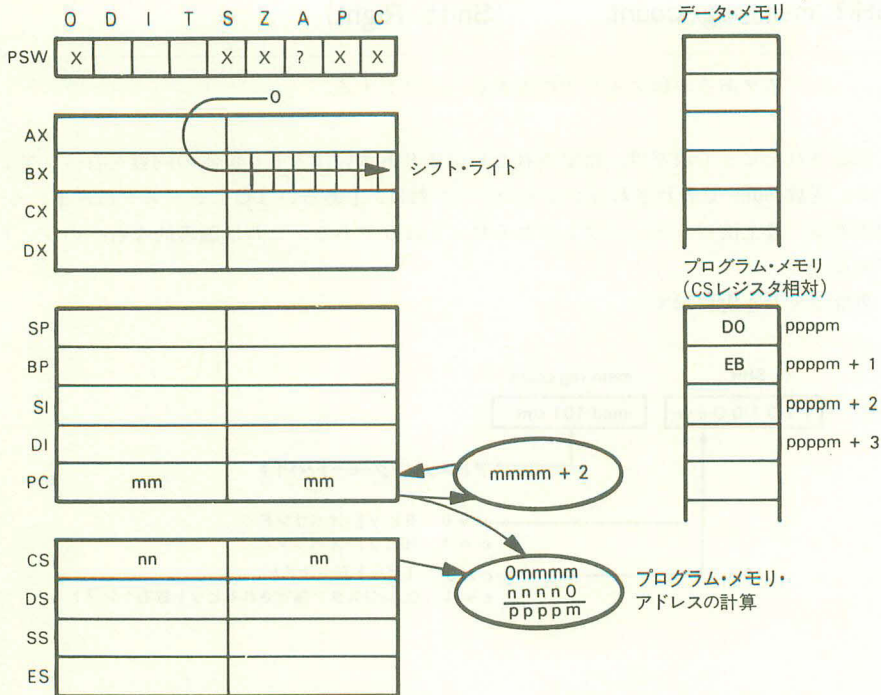
SHR　mem/reg, count　　　(Shift　Right)

レジスタあるいはメモリの内容を右へシフトする．

指定されたビット数だけ，指定されたレジスタあるいはメモリ位置の内容を右へシフトする．変数count で表わされるシフトのビット数は，1あるいはCLレジスタに含まれる数である．最上位ビットにシフトされるビットは0である．これは論理的な右へのシフトである。

命令コードを次に示す．

BLレジスタが$F0_{16}$を含むとする．

SHR　BL

の実行後，BLレジスタの内容は78_{16}になる．

SHR　BL

サイクル数:　レジスタ(1ビットのシフト): 2
　　　　　　レジスタ(Nビットのシフト): 8 + (4 * N)
　　　　　　メモリ　(1ビットのシフト): 15 + EA
　　　　　　メモリ　(Nビットのシフト): 20 + EA + (4*N)

注)

1．算術的な右へのシフトに対して，これは論理的な右へのシフトである．その違いを以下に示す．

論理的右へのシフト（SHR）

すべてのビットを右へ一度にシフトする．最上位ビットには0をシフトさせる．複数ビットのシフトならば，必要なだけ0をシフトさせることを続ける．

算術的右へのシフト（SAR）

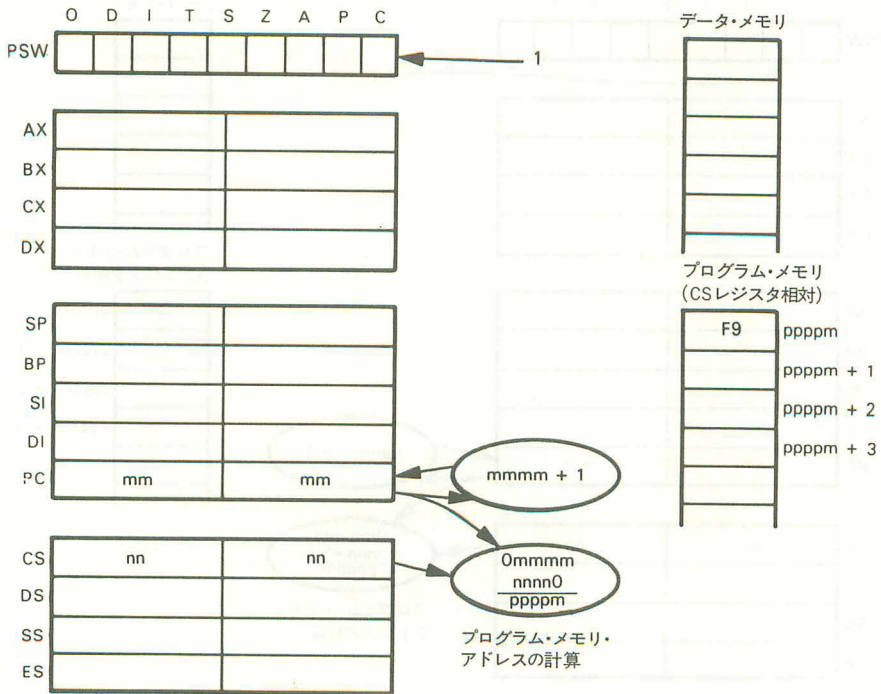
すべてのビットを右へ一度にシフトする．最上位ビットは同じ状態のままにしておく．これは，最上位ビットの符号拡張の意味を持つ．複数ビットのシフトならば，必要なだけ最上位ビットを符号拡張する．

STC　　(Set Carry flag)

キャリー・フラグをセットする．

この命令は，キャリー・フラグを1にするために用いられる．他のステータスあるいはレジスタの内容は影響を受けない．

命令コードを次に示す．

STC
F9

STC
サイクル数: 2

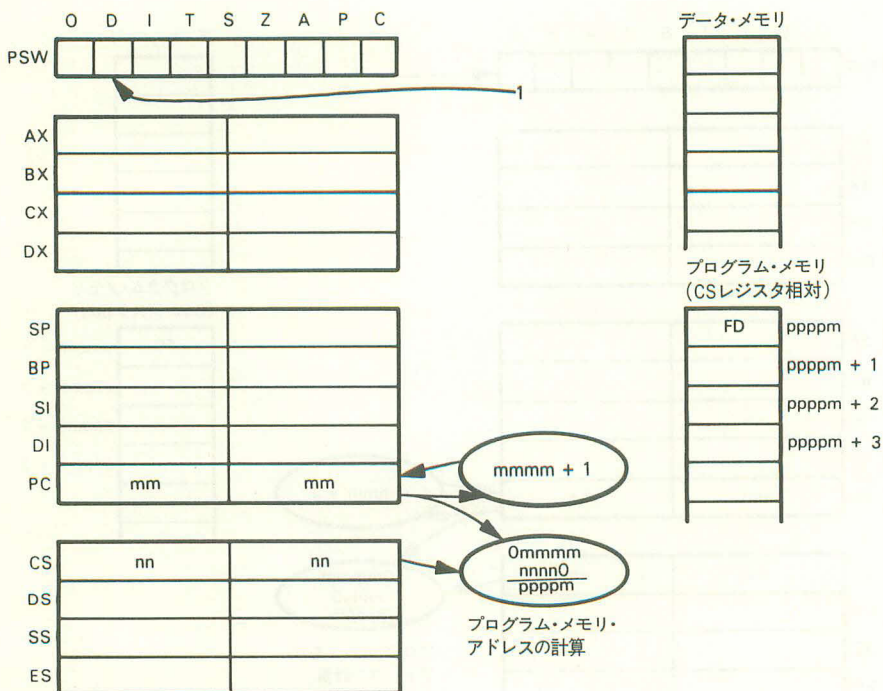
STD (Set Direction flag)

ディレクション・フラグをセットする.

この命令は，ディレクション・フラグを1にするために用いられる．他のステータスあるいはレジスタの内容は影響を受けない．この命令は，ストリング操作で，用いられているポインタを自動減少にする．

命令コードを次に示す．

STD

FD

STD
サイクル数: 2

STI　　　　(Set Interrupt flag)

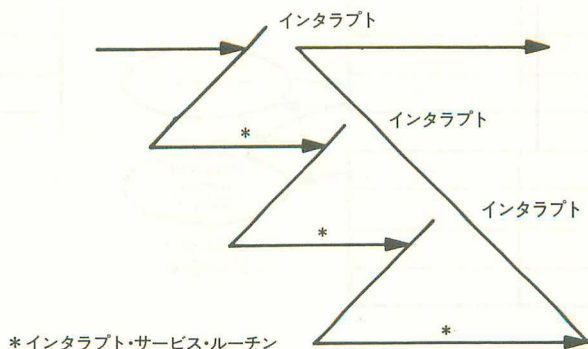
インタラプト・フラグをセットする.

後続の命令実行後に，インタラプト・フラグを1にする．これはインタラプトを可能にする効果を持つ.

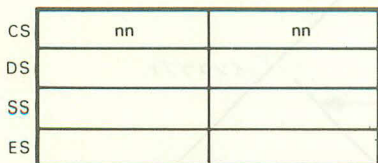
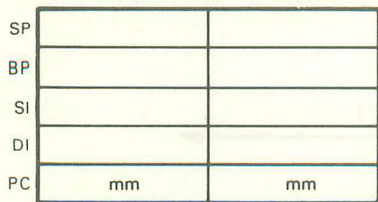
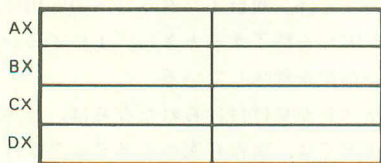
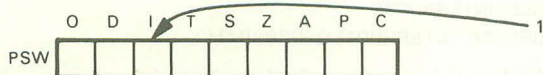
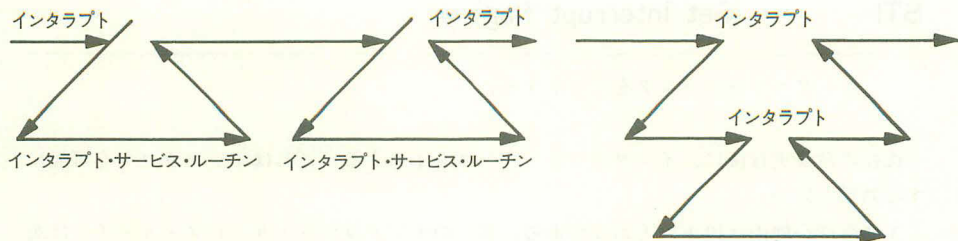
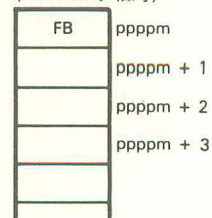
1命令待つ理由は以下のとおりである．多くのインタラプト・サービス・ルーチンは次の2つの命令で終了する.

```
STI      ;ENABLE INTERRUPTS
RET      ;RETURN TO INTERRUPTED PROGRAM
```

もしインタラプトが連続的に処理されるならば，インタラプト・サービス・ルーチンの全期間，すべてのインタラプトは無効となる．このことは，複数インタラプトの応用において，どれかインタラプト・サービス・ルーチンが実行を終了するときに，1つあるいは複数のインタラプトが未処理になる重大な可能性の存在を意味している.

もしＳＴＩ命令が実行されるとただちにインタラプトが受け付けられたならば，リターン命令は実行されないかもしれない．このような状況では，次から次へとスタックが行なわれ，不必要にスタック・メモリ領域を消費する．これは以下のように図示できる.

ＳＴＩに続くさらに1つの命令実行の間インタラプトを禁止することによって，8086ＣＰＵはＲＥＴ命令が続いて実行されることを確実にする。

```
STI      ;ENABLE INTERRUPTS
RET      ;RETURN FROM INTERRUPT
```

インタラプト・サービス・ルーチンを実行している間，インタラプトを無効にしておくことは珍しくなく，インタラプトは連続して処理される。

STI
サイクル数: 2

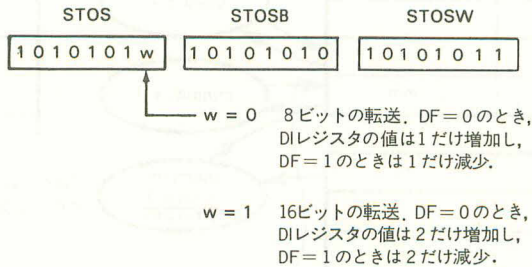
注)

1．この命令は，8080の命令ＥＩと同じ機能を果たす．

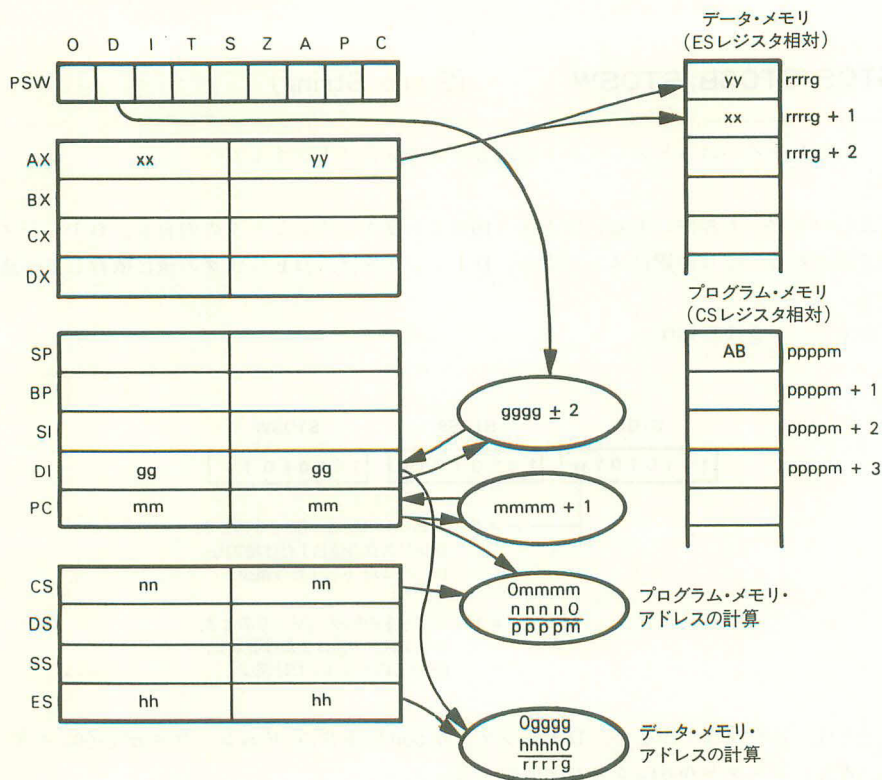
STOS/STOSB/STOSW　(Store String)

ＡＬあるいはＡＸのレジスタの内容をメモリにストアする．

ＡＬ（8ビット操作）あるいはＡＸ（16ビット操作）のレジスタの内容を，ＤＩレジスタで示されるメモリ位置にストアする．ＤＩレジスタは，ＤＦフラグの値に依存して増減する．

命令コードを次に示す．

たとえば，ＤＦフラグが1で，ＤＩレジスタが$000A_{16}$を含み，ＥＳレジスタが2800_{16}を含み，ＡＸレジスタが0604_{16}を含むと仮定する．

STOW

の実行後，メモリ位置$2800A_{16}$のワードの内容は0604_{16}になり，ＤＩレジスタは0008_{16}になる．

STOSW
サイクル数：11：1度だけ実行のとき
9＋(10＊R)：REPプレフィックスによってR回実行したとき

注)

1．ステータスは影響を受けない．

2．この命令のセグメント・アドレスは，常にＥＳレジスタに含まれる．この命令には，セグメント変更プレフィックスは使用できない．もしセグメント変更プレフィックスがあっても無視される．

3．この命令には，ＲＥＰプレフィックスやＬＯＣＫプレフィックスが先行できる．ＲＥＰとＬＯＣＫプレフィックスをこの命令と共に用いることは問題となる．この潜在的な問題の完全な解説は次章を参照．

4．この命令は，バッファあるいはデータ領域の全体を特定の値に設定するのに非常に有用である．以下の一連の命令を考える．

```
LES     DI,JOB$COSTING$ARRAY
MOV     CX,JOB$COSTING$ARRAY$WORD$LENGTH
MOV     AX,0000H
```

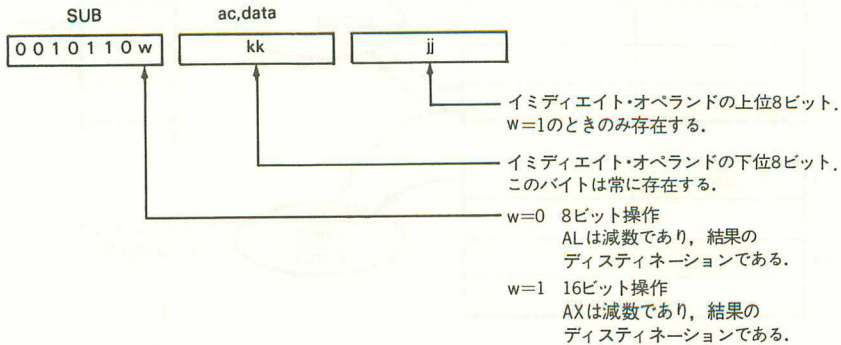
```
REP
STOS WORD
```

この命令の実行後，JOB$COSTING$ARRAYはすべて0を含む．

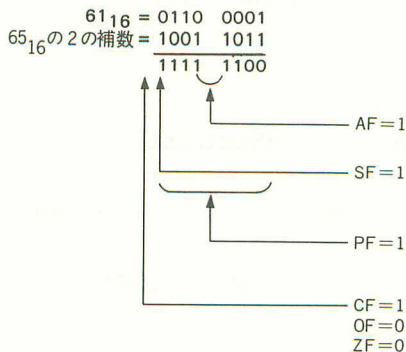
5．STOSの汎用形に対して，8あるいは16ビットのどちらをストアするかをアセンブラがどのように決めるかについての解説は，この章の最後の節を参照．

SUB　ac,data　　　　(Subtract)

ALあるいはAXのレジスタからイミディエイト・データを減じる．

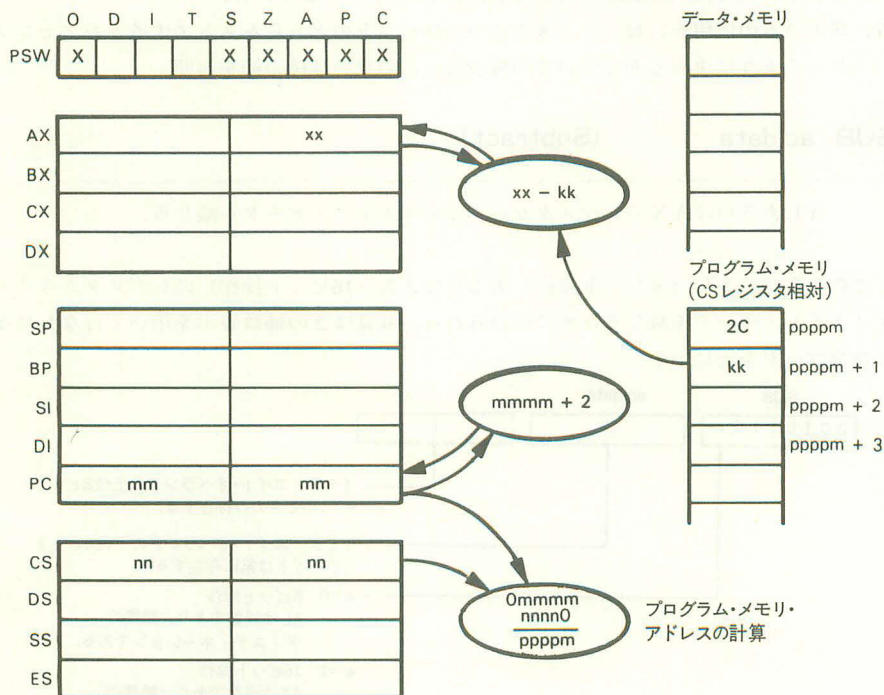
この命令は，AL（8ビット操作）あるいはAX（16ビット操作）のレジスタからイミディエイト・データを減じるために用いられる．減算は2の補数表示を用いて行なわれる．命令コードを次に示す．

たとえば，ALレジスタが61_{16}を含むと仮定する．

SUB　AL,65H

の実行後，ALレジスタの内容はFC_{16}になる．

結果のキャリーは補数がとられることに注意.

FC_{16}は-4の2の補数表示であり，これは実際61_{16}から65_{13}を引いたときの結果になっていることに注意.

SUB AL,kk
サイクル数: 4

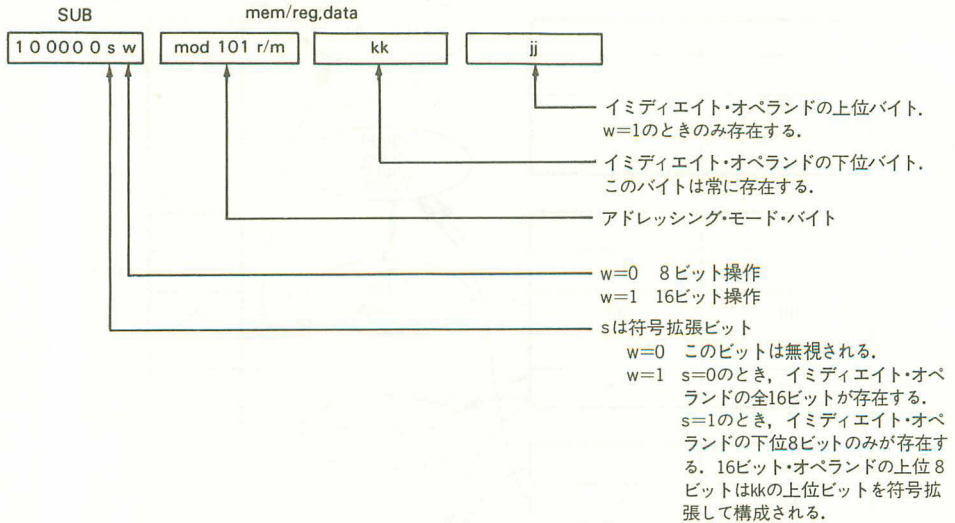
注)

1. この命令は，8080の命令 SUI data と同じ機能を果たす．さらに，16ビットの操作も可能である.

SUB mem/reg,data (Subtract)

レジスタあるいはメモリからイミディエイト・データを減じる.

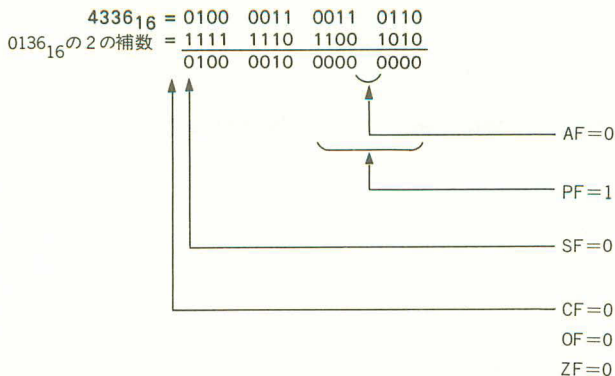
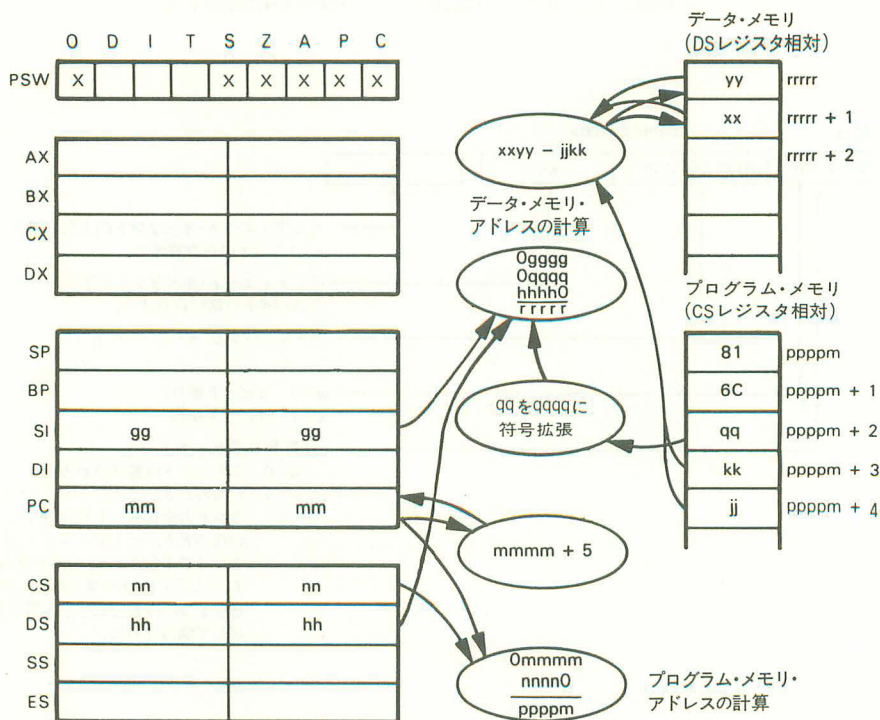
指定されたレジスタあるいはメモリ位置から，後続のプログラム・メモリ・バイトのイ

ミディエイト・データを減じる．8あるいは16ビットの操作が指定できる．

命令コードを次に示す．

DSレジスタが3000_{16}を含み，SIレジスタが0040_{16}を含み，メモリ位置30054_{16}のワードが4336_{16}であると仮定する。

SUB　〔SI＋14H〕,0136H

の実行後，メモリ位置30054_{16}のワードは4200_{16}になる．

フラグは以下のように設定される．

SUB [SI + qq], jjkk
サイクル数：メモリからのイミディエイト減算：17＋EA
レジスタからのイミディエイト減算：4

注)

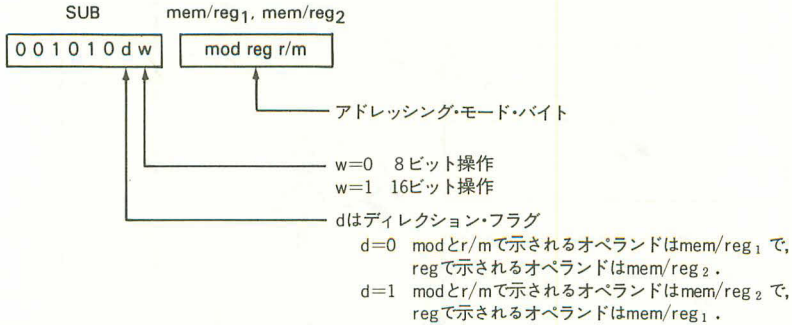
1．この命令は，普通ＡＸあるいはＡＬレジスタからイミディエイト・データを減じるためには用いられない．この目的のためには，命令　ＳＵＢ ac,data がある．

SUB mem/reg$_1$, mem/reg$_2$ (Subtract)

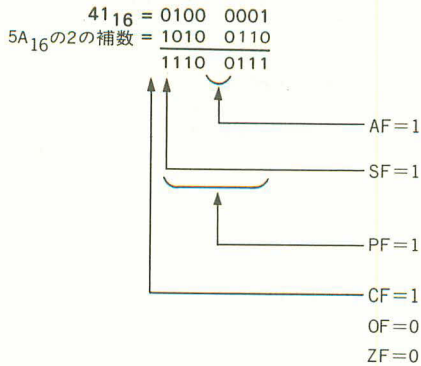
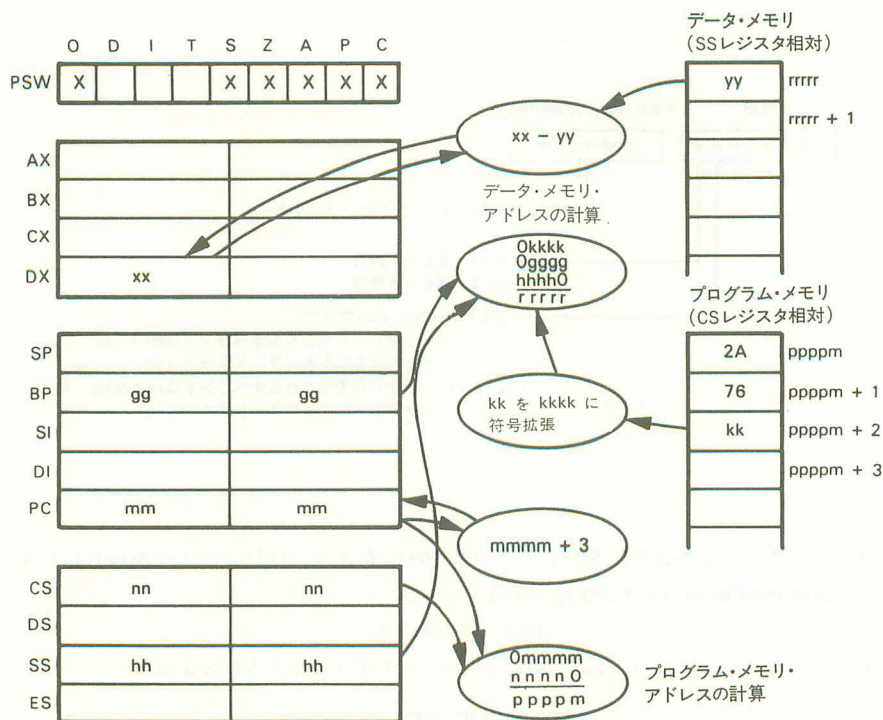
{ レジスタからレジスタ / メモリからレジスタ / レジスタからメモリ } を減じる．

mem/reg$_1$で示されるレジスタあるいはメモリ位置の内容から，mem/reg$_2$で示されるレジスタあるいはメモリ位置の内容を減じる．８あるいは16ビットの操作が指定できる．mem/reg$_1$あるいはmem/reg$_2$はメモリ・オペランドとなるが，オペランドの一方はレジス

タ・オペランドでなければならない．

命令コードを次に示す．

DHレジスタが41_{16}を含み，SSレジスタが0000_{16}を含み，BPレジスタが$00E4_{16}$を含み，メモリ位置$000E8_{16}$のバイトが$5A_{16}$であるとする．

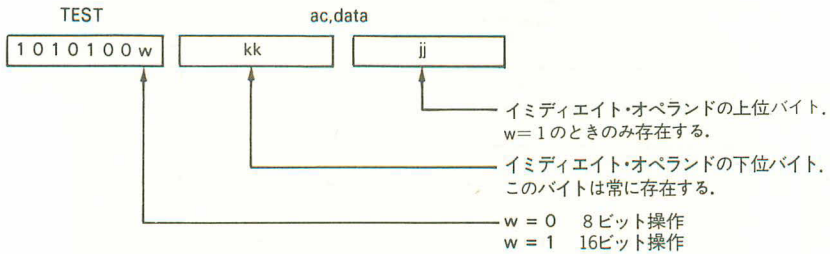
SUB　DH,〔BP+4〕

の実行後，DHレジスタは$E7_{16}$になり，ステータスは以下のように設定される．

SUB DH,[BP + kk]
サイクル数: レジスタからメモリを減算 : 9 + EA
メモリからレジスタを減算 : 16 + EA
レジスタからレジスタを減算: 3

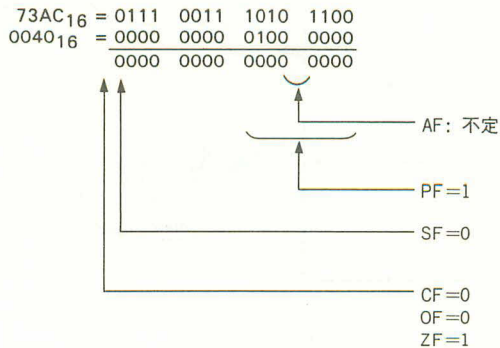
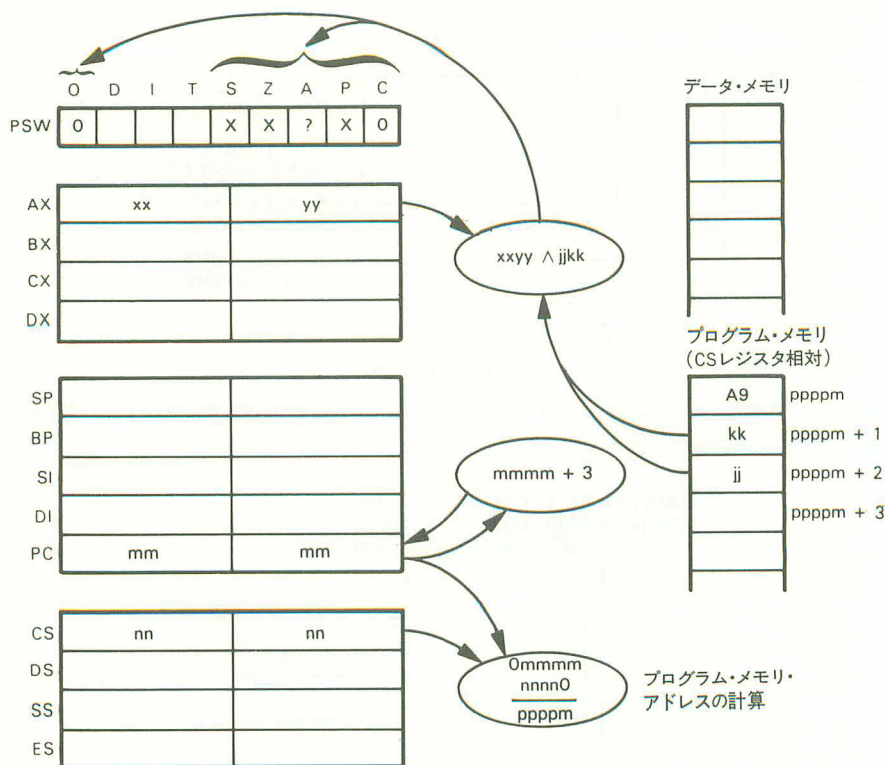
TEST ac, data (Test)

ＡＸあるいはＡＬレジスタとイミディエイト・データをテストする．

ＡＬ（8ビット操作）あるいはＡＸ（16ビット操作）のレジスタの内容と，後続のプログラム・メモリ・バイトのイミディエイト・データとのＡＮＤをとる．ただし，結果はレジスタに返されない．

命令コードを次に示す．

例として，AXレジスタが$73AC_{16}$を含む場合を考える．

TEST　AX,0040H

の実行後，AXレジスタは$73AC_{16}$のままであるが，フラグ・レジスタは$73AC_{16}$と0040_{16}のANDによって変化している．

TEST　AX,jjkk
サイクル数：4

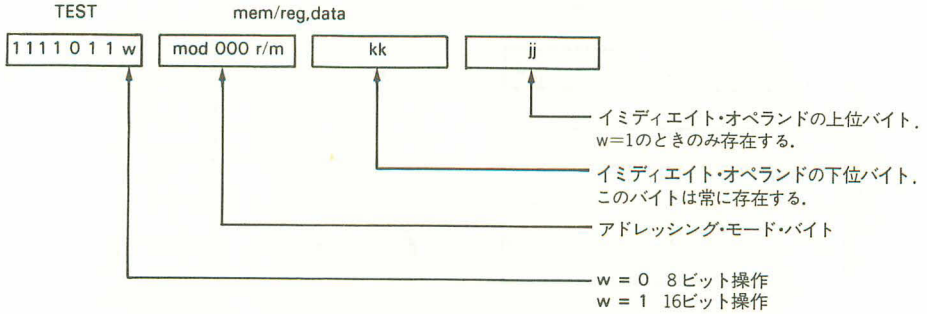
注)

1．他のレジスタあるいはメモリ位置の内容のＴＥＳＴが必要ならば，ＴＥＳＴ mem/reg, data の命令を参照．

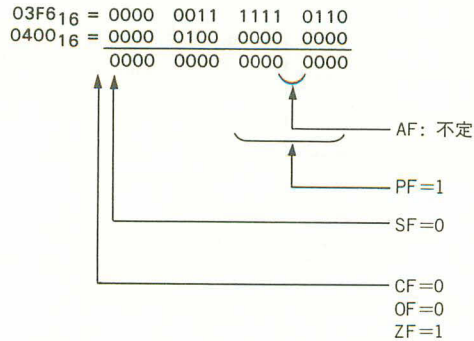
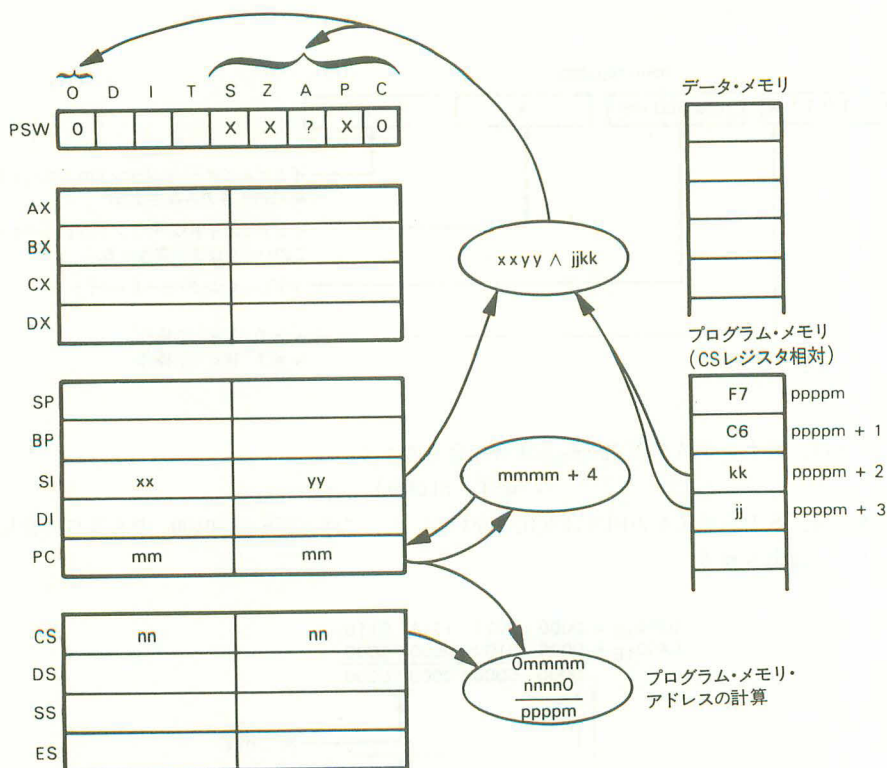
TEST　mem/reg, data　　　(Test)

レジスタあるいはメモリの内容とイミディエイト・データをテストする．

指定されたレジスタあるいはメモリ位置の内容と，後続のプログラム・メモリ・バイトのイミディエイト・データとのＡＮＤをとるが，結果はレジスタあるいはメモリに返されない．8あるいは16ビットの操作が指定できる．

命令コードを次に示す．

たとえば，ＳＩレジスタが$03F6_{16}$を含む場合を考える．

TEST　SI,0400H

の実行後，ＳＩレジスタの内容は変化しないが，フラグは$03F6_{16}$と0400_{16}のＡＮＤの結果によって設定される．

TEST SI,jjkk
サイクル数: メモリとのイミディエイト: 5
レジスタとのイミディエイト: 11 + EA

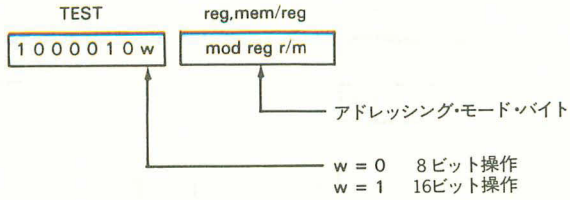
注)

1. ＡＸあるいはＡＬのレジスタのテストの目的のためには ＴＥＳＴ ac,data があるので，それは通常この命令に関連した機能ではない．

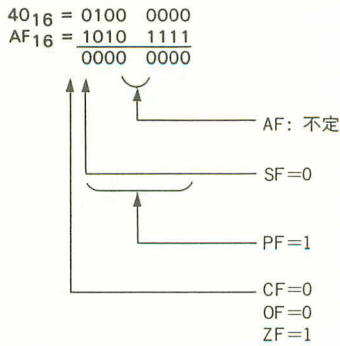
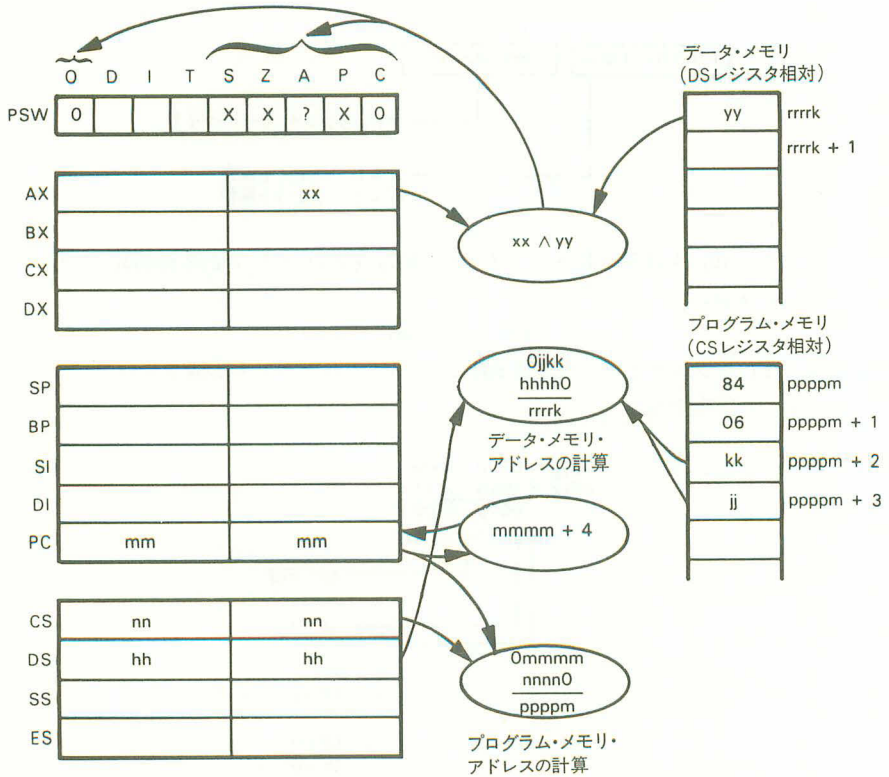
TEST reg, mem/reg (Test)

メモリとレジスタをテストする．

指定されたレジスタあるいはメモリ位置の内容と，指定されたレジスタの内容とのＡＮＤをとり，その結果をフラグに設定するが，結果はレジスタあるいはメモリに返されない．8あるいは16ビットの操作が指定できる．

命令コードを次に示す．

ＡＬレジスタが 40_{16} を含み，ＤＳレジスタが 8800_{16} を含み，メモリ位置 88053_{16} のバイトが AF_{16} であるとする．

TEST　AL,〔53H〕

の実行後，ＡＬレジスタとメモリ位置 88053_{16} のバイトはどちらも影響を受けないが，フラグは次のように変化する．

TEST　AL,[kk]

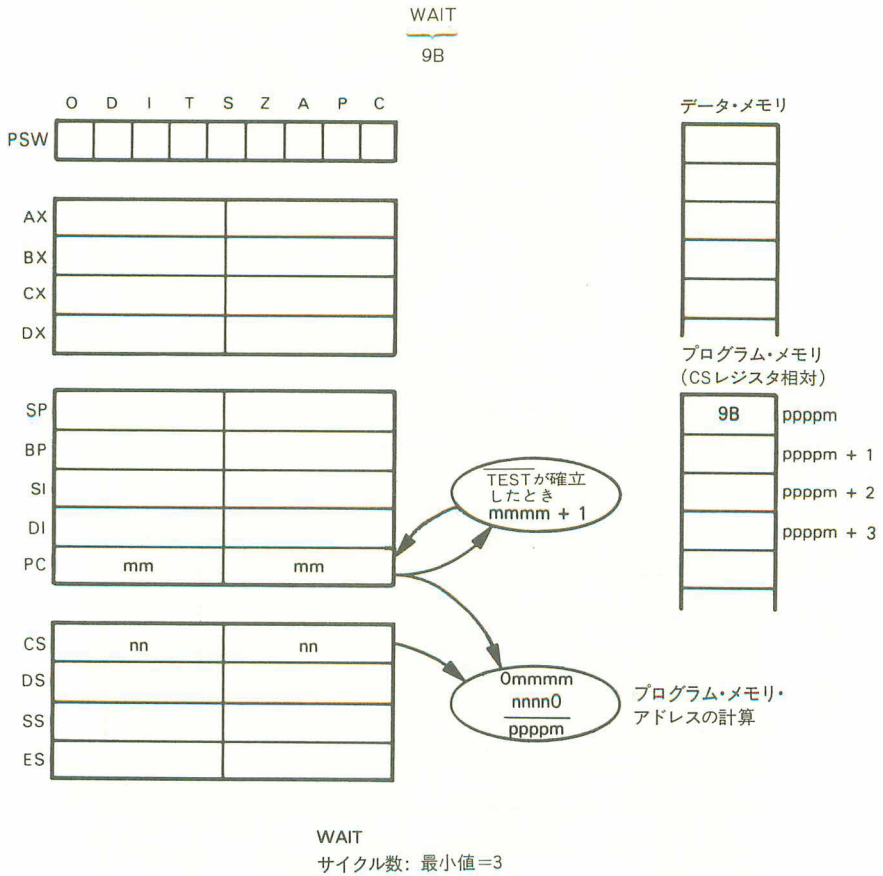
サイクル数：レジスタとメモリ：9 + EA
　　　　　　レジスタとレジスタ：3

WAIT　　　(Wait)

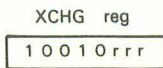
テスト・ピンの信号が確立するのを待つ．

$\overline{\text{TEST}}$ピンの信号が確立していなければ，この命令によって8086はアイドル状態となる．8086がアイドル状態から抜け出すのは，以下の２つの条件の１つが成立したときである．

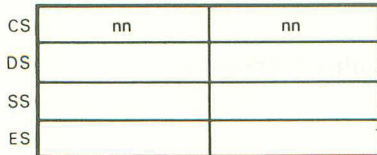
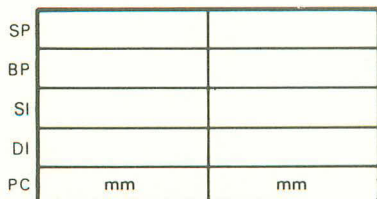
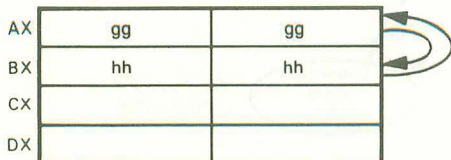
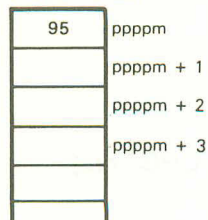
1．インタラプトが有効ならば，外部インタラプトによって8086はインタラプトの処理を行なう．8086がインタラプトの処理を行なうときにセーブされるアドレスは，ＷＡＩＴ命令のアドレスである．したがって，インタラプト・サービス・ルーチンから復帰すると，ＷＡＩＴ命令に戻る．
2．$\overline{\text{TEST}}$信号が確立している．

命令コードを次に示す．

XCHG reg　　　　(Exchange)

アキュムレータとレジスタの内容を交換する．

アキュムレータの内容と，指定された16ビットの内容を交換する．
命令コードを次に示す．

たとえば，

XCHG　BX

の命令は，ＢＸレジスタの内容をＡＸレジスタの内容と交換するために用いられる．

XCHG　BX
サイクル数：3

注）

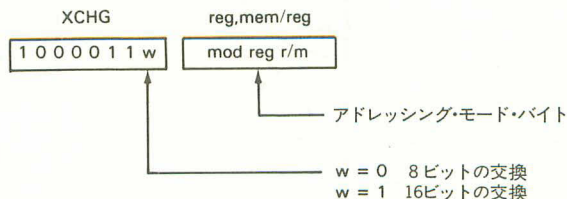
1．ステータスは影響を受けない．

2．命令　XCHG　AX　は，8086で　NOP　命令として用いられる．

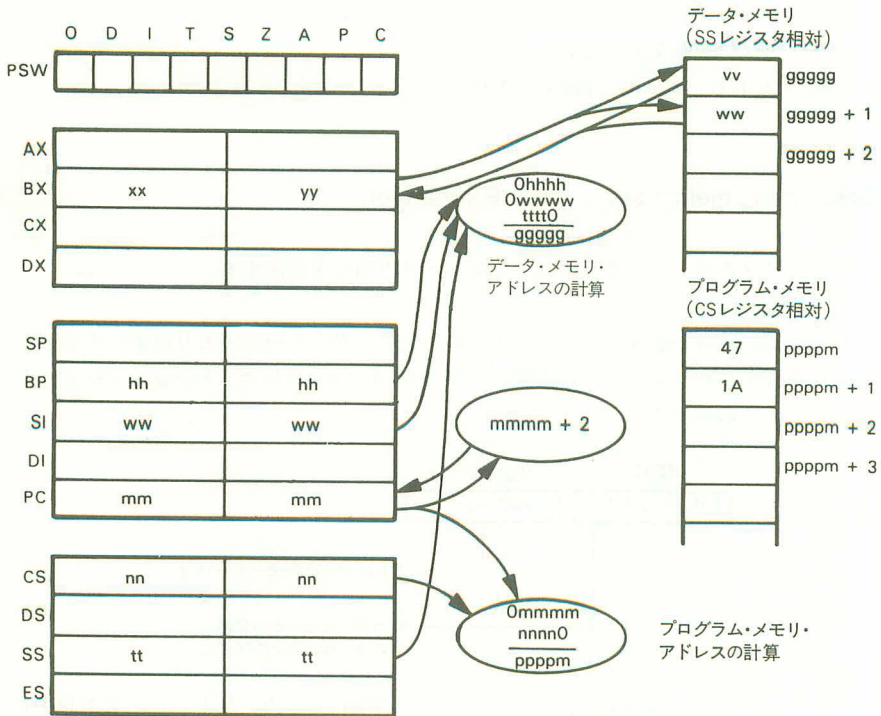
XCHG　reg, mem/reg　　(Exchange)

レジスタあるいはメモリの内容とレジスタの内容を交換する．

この命令は，mem/reg オペランドで示されるレジスタあるいはメモリ位置の内容を，reg オペランドで示されるレジスタの内容と交換する．8あるいは16ビットの転送が指定できる．

命令コードを次に示す．

BXレジスタの内容が$6F30_{16}$で，SSレジスタが$2F00_{16}$を含み，SIレジスタが0046_{16}を含み，BPレジスタが0200_{16}を含み，メモリ位置$2F246_{16}$のワードが　4154_{16}の場合を考える．

XCHG　BX,〔BP+SI〕

の実行後，BXレジスタは4154_{16}になり，メモリ位置$2F246_{13}$は30_{16}に，メモリ位置$2F247_{16}$は$6F_{16}$になる．

XCHG　BX, [BP + SI]
サイクル数：メモリとレジスタ：17 + EA
　　　　　　レジスタとレジスタ：4

注)

1．ステータスは影響を受けない．

2．セグメント・レジスタはこの命令で指定できない．セグメント・レジスタの内容を交換する命令は存在しない．

3．普通，この命令はレジスタとAXレジスタの交換には用いられない．このためには，命令 XCHG reg がある．

XLAT　　　(Translate)

ALとBXのレジスタによってテーブルを検索する．

8ビットのデータがALレジスタにロードされる．このデータ要素のアドレスは，次のアルゴリズムを用いて求められる．

1．16ビットのBXレジスタに，ALレジスタの8ビットの内容を加算する．

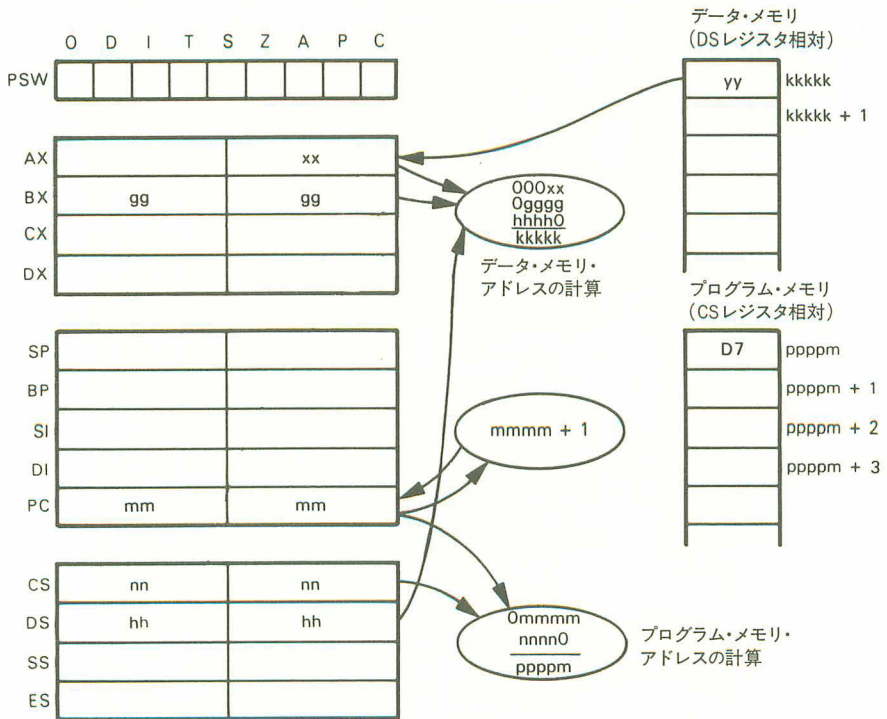
2．ステップ1の加算結果を，DSレジスタに対するオフセット・アドレスとして用いる（セグメント変更が行なわれていないと仮定）．

XLATの命令コードを次に示す．

XLAT
D7

たとえば，ALレジスタが$0F_{16}$を含み，BXレジスタが0040_{16}，DSレジスタが$F000_{16}$とすると，

XLAT

の実行によって，メモリ位置$F004F_{16}$の内容がALレジスタにロードされる．

XLAT
サイクル数：11

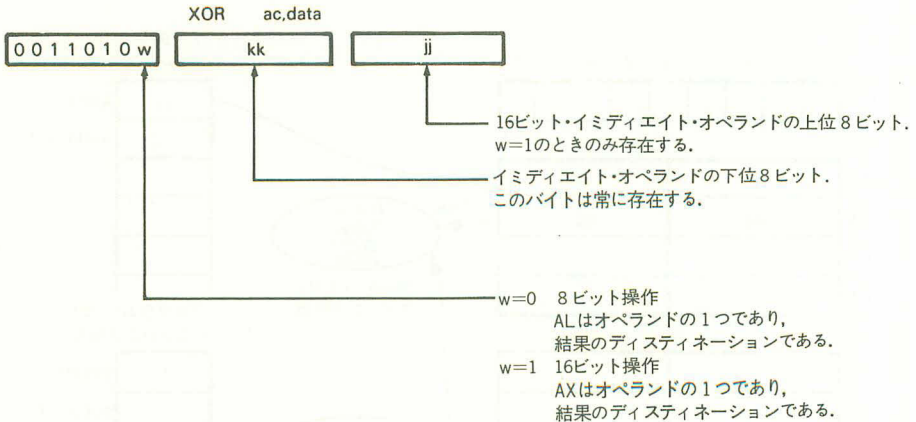
注）

1．この命令は，BXレジスタがテーブルの開始アドレスを含み，ALレジスタをテーブルのインデックスとして利用する場合に用いられるのが最も一般的である．

XOR　ac, data　　　(Exclusive-OR)

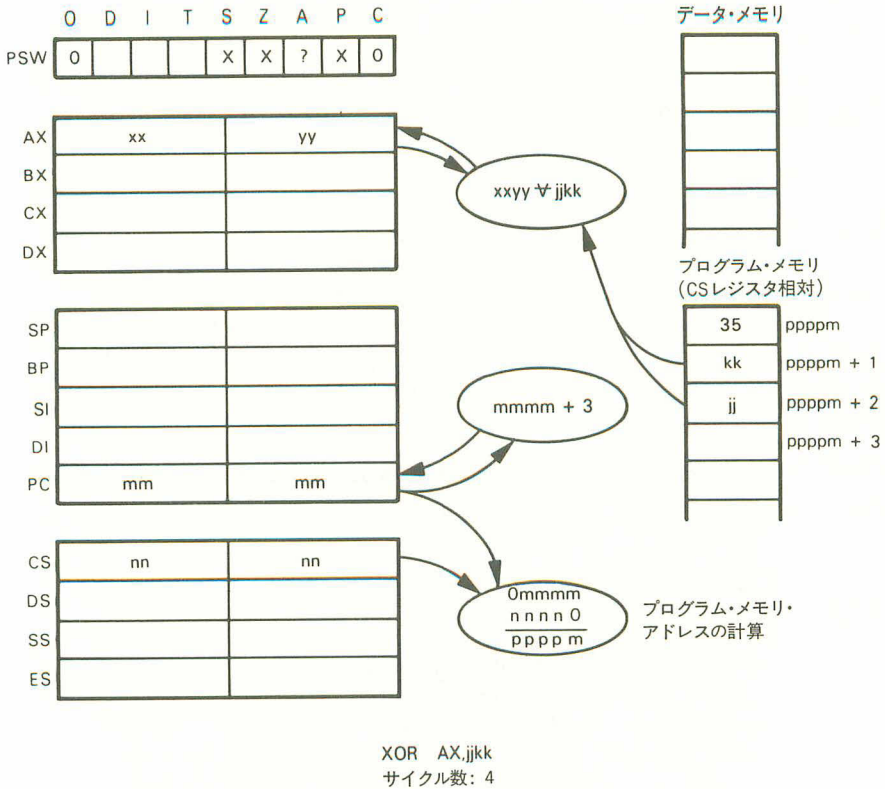
AXあるいはALのレジスタとイミディエイト・データのXORをとる.

この命令は，イミディエイト・アドレッシングによって，AL（8ビット）あるいはAX（16ビット）のレジスタと，8あるいは16ビットのデータとのエクスクルーシブORをとる.

命令コードを次に示す.

たとえば，AXが $B31C_{16}$ を含むとする.

XOR　AX, 5522H

の実行によって，AXレジスタには $E63E_{16}$ がストアされる.

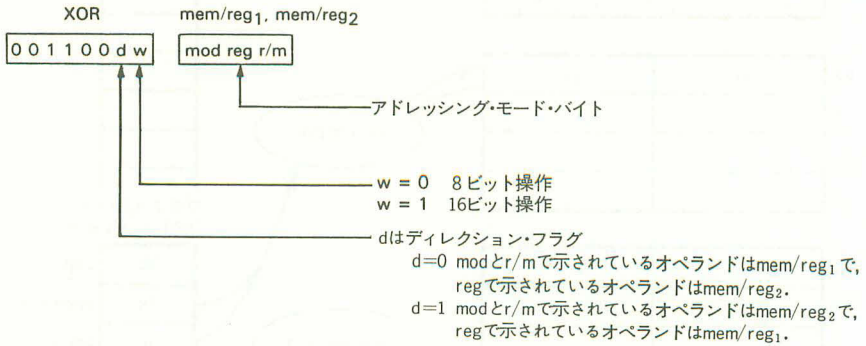
XOR　AX,jjkk
サイクル数：4

注)

1．この命令は，8080アセンブリ言語において，ＸＲＩ data 命令と同じ機能を果たす．ただし，この命令は16ビットのデータも可能であるが，8080のＸＲＩは8ビットのデータ要素のみを用いる．

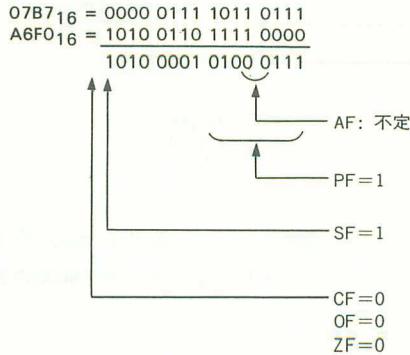
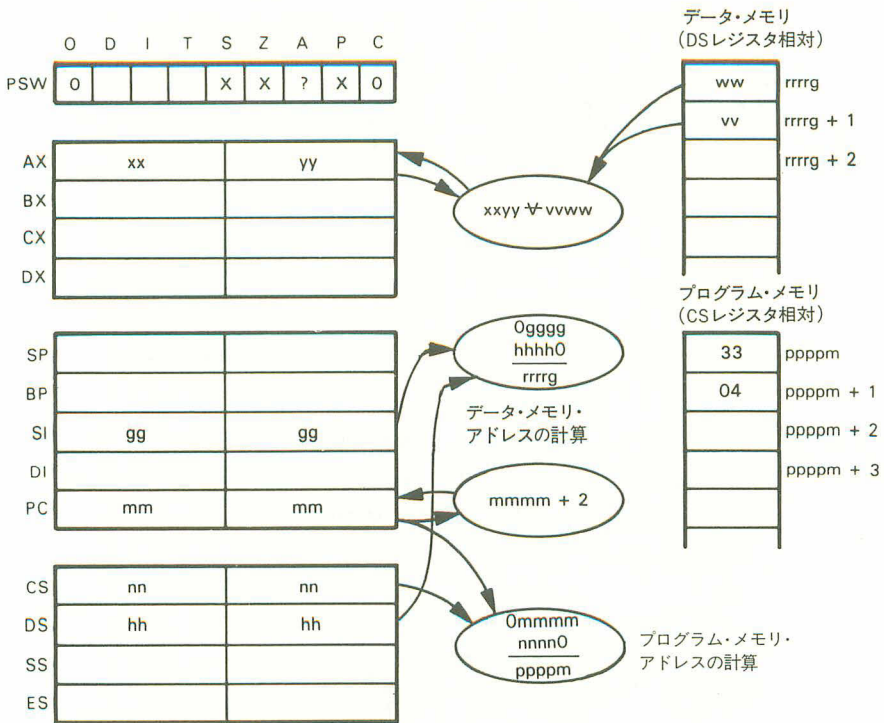
XOR　mem/reg_1, mem/reg_2　　(Exclusive-OR)

$\left\{\begin{array}{l}\text{レジスタとレジスタ}\\\text{レジスタとメモリ}\\\text{メモリとレジスタ}\end{array}\right\}$ でＸＯＲをとる．

mem/reg_2で示されるレジスタあるいはメモリ位置の内容と，mem/reg_1で示されるレジスタあるいはメモリ位置の内容のエクスクルーシブＯＲをとり，結果をmem/reg_1に返す．8あるいは16ビットの操作が指定できる．mem/reg_1あるいはmem/reg_2はメモリ・オペランドとなるが，オペランドの一方はレジスタ・オペランドでなければならない．

命令コードを次に示す．

AＸレジスタが $07B7_{16}$ を含み，ＤＳレジスタが 9080_{16} を含み，ＳＩレジスタが $040E_{16}$ を含み，メモリ位置 $90C0E_{16}$ のワードが $A6F0_{16}$ であるとする．

XOR　AX,〔SI〕

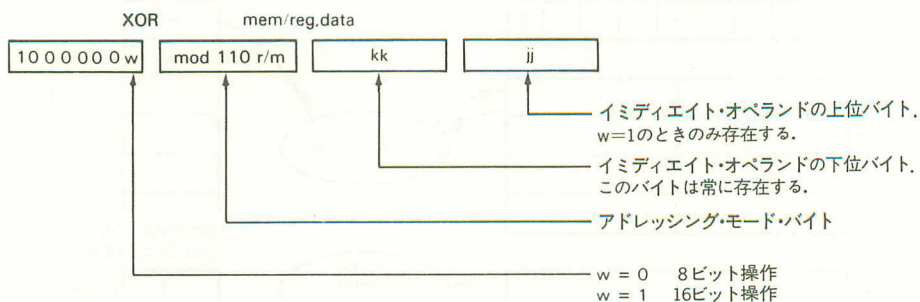
の実行後，AＸレジスタは $A147_{16}$ になる．フラグは以下のように設定される．

XOR　AX,[SI]

サイクル数：メモリからレジスタに対して：9 + EA
　　　　　　レジスタからメモリに対して：16 + EA
　　　　　　レジスタからレジスタに対して：3

XOR　mem/reg, data　(Exclusive-OR)

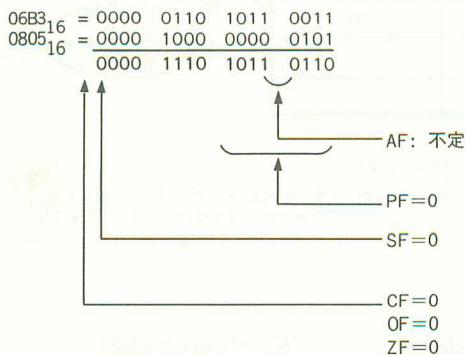
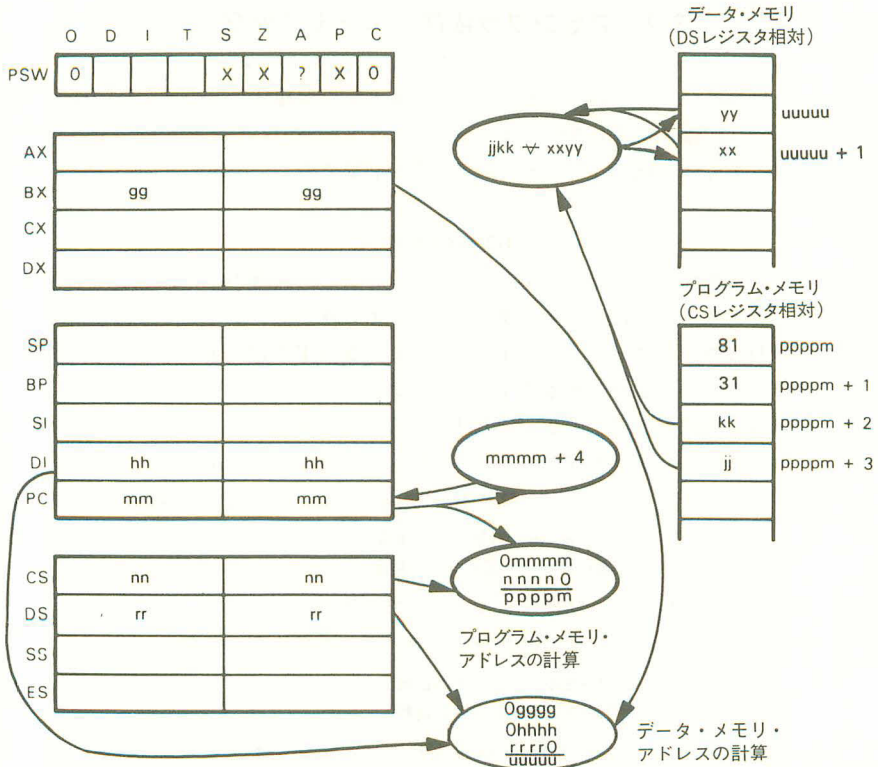
レジスタあるいはメモリの内容とイミディエイト・データのXORをとる．

指定されたレジスタあるいはメモリ位置の内容と，後続のプログラム・メモリ・バイトのイミディエイト・データとのXORをとる．8あるいは16ビットの操作が指定できる．

命令コードを次に示す．

ＤＳレジスタが3800_{16}を含み，ＢＸレジスタの内容が0200_{16}で，ＤＩレジスタが0136_{16}を含み，メモリ位置38336_{16}のワードが$06B3_{16}$の場合を考える．

XOR　〔BX+DI〕, 0805H

の実行後，メモリ位置38336_{16}のワードは$0EB6_{16}$になる．

注）

1．この命令は，通常AXあるいはALのレジスタとイミディエイト・データのXORをとるためには用いられない．この目的のためには，命令　XOR　ac, data　がある．

3.7 アセンブラ依存のニーモニック

8086アセンブラのいくつかのニーモニックは，8あるいは16ビットのいずれの操作が行なわれるべきかは明確に定義しない．オペランドとして8086のレジスタを持つすべての命令は，8ビットあるいは16ビットのどちらの操作が必要であるかを決めるのに，レジスタを用いることができる．たとえば，

```
XOR AX,0804H
```

の命令が16ビット操作であることは明らかである．しかし，命令の中にはレジスタを指定しないが8ビットあるいは16ビットの操作が可能なものがある．このような命令には，ＣＭＰＳやＬＯＤＳなどのストリング操作と，ＭＵＬやＮＯＴなどの単一メモリ・オペランドが指定できる命令の，2つの基本的な形式が含まれている．

アセンブラはこの問題を以下の3つの方法で処理している．

1．ワードあるいはバイトの操作のどちらを指定するかを表わす文字を，ニーモニックに付加することができる．たとえば，ＭＵＬは次のように指定できる．

```
MULB  8×8ビットの乗算
MULW  16×16ビットの乗算
```

ストリング操作の場合，文字はニーモニックの最後のＢ（バイト）あるいはＷ（ワード）である．たとえば，命令ＣＭＰＳは次のもので置き換えられる．

```
CMPSB  8ビットの比較
CMPSW  16ビットの比較
```

2．ストリング操作のオペランドは，ＷＯＲＤあるいはＢＹＴＥである．たとえば，ＭＯＶＳ命令は次のようになる．

```
MOVS BYTE  8ビットの移動
MOVS WORD  16ビットの移動
```

さらに，この方法は単一メモリ・オペランドへ適用できる．次に例を示す．

```
NOT  SI,BYTE
NOT  SI,WORD
```

3．プログラマは，すべてのデータ領域とシンボルを，ＷＯＲＤあるいはＢＹＴＥの要素として定義する．アセンブラはこの情報を保持していて，データ領域やシンボルへの参照が行なわれたときに，8あるいは16ビットの操作のどちらが必要かを決定する．たとえば，

```
TOUCH$TONE$OUTPUT$BYTE   DB 00H    ; DB means define byte
TIMER                    DW 0000H  ; DW means define word
```

このとき，以下の操作が行なわれるならば，

```
NOT  TOUCH$TONE$OUTPUT$BYTE
INC  TIMER
```

ＮＯＴ操作は8ビットの操作としてアセンブルされ，ＩＮＣ操作には16ビットの操作

が指定される．特定のアセンブラは，その機能を果たすために，これらのオプションのうちの１つ以上が用いられることに注意．

ストリング命令について，オブジェクト・コード・レベルの命令はオペランドを含んでいないが，命令の汎用のソース形式ではオペランドを指定する必要がある．この条件は，アセンブラにバイトあるいはワードのどちらの操作が必要かの決定を可能とする．たとえば，

```
LODS
```

の命令を考える．ソース・オペランドのアドレスはＤＳとＳＩのレジスタで指定され，ディスティネーションはＡＸあるいはＡＬのレジスタとなる．ニーモニックＬＯＤＳだけでは，オペランドの型（バイトあるいはワード）を決定するのに十分な情報は得られない．しかし，

```
ID_NUMBER    DB 5
                -
                -
                -
             LODS  ID_NUMBER
```

によって，ＬＯＤＳからバイト・ロードの形式をアセンブラが自動的に構成できるデータの型の情報が得られる．この表記法はまた，プログラマがロード内容を指定できるので，命令コードの読みやすさと保守性を高めている．

第4章
8086の命令グループ

この章には，8086命令セットについて別の解説が含まれている．すなわち各命令が個々に述べられている3章の説明に対して，この章では命令のグループについて述べる．8086の命令は，実行される機能によって次のグループに分かれる．

- データ移動命令
- 算術演算命令
- 論理演算命令
- ストリング・プリミティブ命令
- プログラム・カウンタ制御命令
- プロセッサ制御命令
- 入出力命令
- インタラプト命令
- ローテートとシフトの命令

4.1　データ移動命令

データの移動を行なう8086の命令を**表4-1**に示す．8086のデータ移動命令は，以下の3つの一般的部類に分けられる．

1. レジスタからレジスタへ，あるいはメモリ位置とレジスタの間で，データを移動する命令．
2. スタックに対して，データを移動する命令．
3. あるメモリ位置から他へ，複数のバイトを移動する命令．

この第1と第2の命令のタイプについては，この節で述べる．ストリング・プリミティブと呼ばれる命令を用いて構成される複数バイトの移動命令については，この節では外面だけを述べる．その詳細は本章の後で述べる．

データ移動命令は，次のタイプのルーチンで用いられる．

表4-1　8086のデータ移動命令

ニーモニック	オペランド	オブジェクト・コード	バイト	クロック	ステータス									動作内容
					O	D	I	T	S	Z	A	P	C	
MOV	mem/reg$_1$, mem/reg$_2$	100010dw mod rrr r/m (DISP) (DISP)	2, 3または4	reg - reg: 2 mem - reg: 8 + EA reg - mem: 9 + EA										〔mem/reg$_1$〕←〔mem/reg$_2$〕* 8あるいは16ビットのデータ要素を，mem/reg$_2$で示されるメモリ位置あるいはレジスタから，mem/reg$_1$で示されるメモリ位置あるいはレジスタに移動する．
MOV	mem/reg, data	1100011w mod 000 r/m (DISP) (DISP) kk jj (w=1のとき)	3, 4, 5または6	10 + EA										〔mem/reg〕←data 8あるいは16ビットのイミディエイト・データを，mem/regで示されるメモリ位置あるいはレジスタに移動する．
MOV	reg,data	1011wrrr kk jj (w=1のとき)	2または3	4										〔reg〕←data 8あるいは16ビットのイミディエイト・データを，regで示されるレジスタに移動する．
MOV	ac,mem	1010000w kk jj	3	10										〔ac〕←〔mem〕 memで示されるメモリ位置からデータを，AL（8ビット操作）あるいはAX（16ビット操作）のレジスタに移動する．
MOV	mem,ac	1010001w kk jj	3	10										〔mem〕←〔ac〕 AL（8ビット操作）あるいはAX（16ビット操作）のレジスタからデータを，memで示されるメモリ位置に移動する．
MOV	segreg, mem/reg	8E mod 0 ss r/m (DISP) (DISP)	2, 3または4	reg - reg: 2 mem - reg: 8 + EA										〔segreg〕←〔mem/reg〕 mem/regで示されるメモリ位置あるいはレジスタから16ビットのデータを選択されたセグメント・レジスタに移動する．ss=01のときは定義されない．
MOV	mem/reg, segreg	8C mod 0 ss r/m (DISP) (DISP)	2, 3または4	reg - reg: 2 mem - reg: 9 + EA										〔mem/reg〕←〔segreg〕 選ばれたセグメント・レジスタの内容を，指定されたメモリ位置あるいはレジスタに移動する．
XCHG	reg, mem/reg	1000011w mod rrr r/m (DISP) (DISP)	2, 3または4	reg - reg: 4 reg - mem: 17 + EA										〔reg〕←→〔mem/reg〕 regで示されるレジスタの8あるいは16ビットの内容を，mem/regで示されるメモリ位置あるいはレジスタの内容と交換する．

*mem←memは含まれない．

表4-1　8086のデータ移動命令（続き）

ニーモニック	オペランド	オブジェクト・コード	バイト	クロック	ステータス									動作内容
					O	D	I	T	S	Z	A	P	C	
XCHG	reg	10010rrr	1	3										〔AX〕←→〔reg〕 AXレジスタの内容と指定されたレジスタの内容とを交換する．
XLAT		D7	1	11										〔AL〕←〔〔AL〕＋〔BX〕〕 ALとBXの和で示されるデータ・バイトをALレジスタにロードする．
LDS	reg,mem	C5 mod rrr r/m (DISP) (DISP)	2，3または4	16 + EA										〔reg〕←〔mem〕，〔DS〕←〔mem＋2〕 memで示されるメモリ位置から16ビットのデータを選択されたレジスタにロードする．memで示されるメモリ位置の次の16ビットのデータをDSレジスタにロードする．
LEA	reg,mem	8D mod rrr r/m (DISP) (DISP)	2，3または4	2 + EA										〔reg〕←mem（アドレスのオフセット部分） メモリ・アドレスのオフセット部分となる16ビットを選択されたレジスタにロードする．
LES	reg,mem	C4 mod rrr r/m (DISP) (DISP)	2，3または4	16 + EA										〔reg〕←〔mem〕，〔ES〕←〔mem＋2〕 memで示されるメモリ位置から16ビットのデータを選択されたレジスタにロードする．memで示されるメモリ位置の次の16ビットのデータをESレジスタにロードする．
PUSH	mem/reg	FF mod 110 r/m (DISP) (DISP)	2，3または4	reg: 11 mem: 16 + EA										〔SP〕←〔SP〕－2，〔〔SP〕〕←〔mem/reg〕 SPを2だけ減じる．mem/regで示されるメモリ位置あるいはレジスタの16ビットの内容をスタックのトップにストアする．
PUSH	reg	01010rrr	1	10										〔SP〕←〔SP〕－2，〔〔SP〕〕←〔reg〕 SPを2だけ減じる．指定されたレジスタの16ビットの内容をスタックのトップにストアする．
PUSH	segreg	000ss110	1	10										〔SP〕←〔SP〕－2，〔〔SP〕〕←〔segreg〕 SPを2だけ減じる．指定されたセグメント・レジスタの16ビットの内容をスタックのトップにストアする．
PUSHF		9C	1	10										〔SP〕←〔SP〕－2，〔〔SP〕〕←〔FLAGS〕 SPを2だけ減じる．フラグ・レジスタの内容をスタックのトップにストアする．

表4-1　8086のデータ移動命令(続き)

ニーモニック	オペランド	オブジェクト・コード	バイト	クロック	ステータス									動　作　内　容
					O	D	I	T	S	Z	A	P	C	
POP	mem/reg	8F mod 000 r/m (DISP) (DISP)	2, 3または4	reg: 8 mem: 17 + EA										〔mem/reg〕←〔〔SP〕〕,〔SP〕←〔SP〕+ 2 スタックのトップの16ビットを，mem/regで示されるメモリ位置あるいはレジスタに移動する．SPを2だけ増やす．
POP	reg	01011rrr	1	8										〔reg〕←〔〔SP〕〕,〔SP〕←〔SP〕+2 スタックのトップの16ビットを，選択されたレジスタに移動する．SPを2だけ増やす．
POP	segreg	000ss111	1	8										〔segreg〕←〔〔SP〕〕,〔SP〕←〔SP〕+ 2 スタックのトップの16ビットを，指定されたセグメント・レジスタに移動する．SPを2だけ増やす． ss=01のときは定義されない．
POPF		9D	1	8	X	X	X	X	X	X	X	X	X	〔FLAGS〕←〔〔SP〕〕,〔SP〕←〔SP〕+ 2 スタックのトップの16ビットをフラグ・レジスタに移動する．SPを2だけ増やす．
LAHF		9F	1	4										8080AのフラグをAHレジスタに移動する． [S Z _ A _ P _ C] → AH
SAHF		9E	1	4					X	X	X	X	X	AHレジスタを8080Aのフラグに移動する． [S Z _ A _ P _ C] ← AH

1．BUFFER$Aの内容をBUFFER$Bに移すルーチン．
2．BUFFER$Aの内容を初期設定するルーチン．
3．BUFFER$Aの内容を変換するルーチン．

4.1.1　バッファからバッファへの移動ルーチン

8ビットと16ビットのデータ要素に対して，2つの基本的なバッファからバッファへの移動ルーチンを，それぞれ，**図4-1**と**図4-2**に示す．このルーチンでは，ＳＩレジスタがBUFFER$Aのアドレスを含み，ＤＩレジスタがBUFFER$Bのアドレスを含み，ＣＸレジスタが移動するデータ要素の数を含むと仮定している．

```
MOVE$BYTES:     MOV   AL,[SI]          ;LOAD BYTE FROM SOURCE
                MOV   [DI],AL          ;STORE BYTE INTO DESTINATION
                INC   SI               ;ADJUST POINTERS
                INC   DI
                DEC   CX               ;DECREMENT # TO MOVE
                JNZ   MOVE$BYTES       ;LOOP IF NOT DONE
                RET
```

図4-1　8ビットのバッファからバッファへの移動

```
MOVE$WORDS:     MOV   AX,[SI]          ;LOAD WORD FROM SOURCE
                MOV   [DI],AX          ;STORE WORD INTO DESTINATION
                                       ;ADJUST POINTERS
                INC   SI
                INC   SI
                INC   DI
                INC   DI
                DEC   CX               ;DECREMENT # TO MOVE
                JNZ   MOVE$WORDS       ;LOOP IF NOT DONE
                RET
```

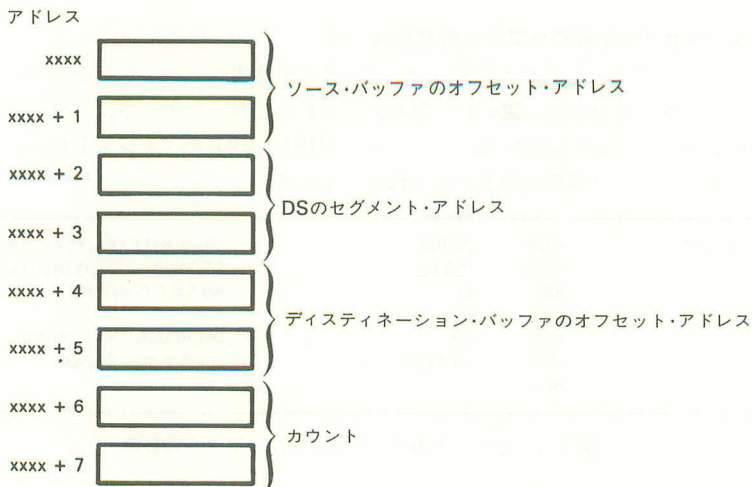
図4-2　16ビットのバッファからバッファへの移動

図4-1と図4-2に示した一連の命令は，データ・セグメント内でデータを移動する．このルーチンでは，ＳＩあるいはＤＩのレジスタの代わりにＢＸレジスタを用いることができる．

ここに示したルーチンは容易に理解できるが，あまり能率的ではない．この章で後程述べるストリング・プリミティブ操作は，より効力のあるバッファからバッファへの移動ルーチンを構成する．また，ＬＯＯＰ命令は，デクリメントと分岐によるプログラム・ロジックを能率的なものにする．

(1)　バッファからバッファへの移動のためのレジスタの初期設定

バッファからバッファへの移動ルーチンなどの処理で用いられるレジスタの初期設定には，多くの方法がある．初期設定の方法は，アドレスとカウントがどのように得られるかにかかっている．付加的要素には，初期設定されるべきレジスタの数がある．たとえば，

多くの場合,ＤＳレジスタは既に初期設定されている.バッファからバッファへの移動ルーチンの,バッファ先頭アドレスとバイトあるいはワードの数は,レジスタで示されるメモリ・ワードのブロックに保持することができる． 次の8バイトのメモリ・ブロックを考える.

上に示されているように，対になるメモリ・バイトはアドレスを保持する.

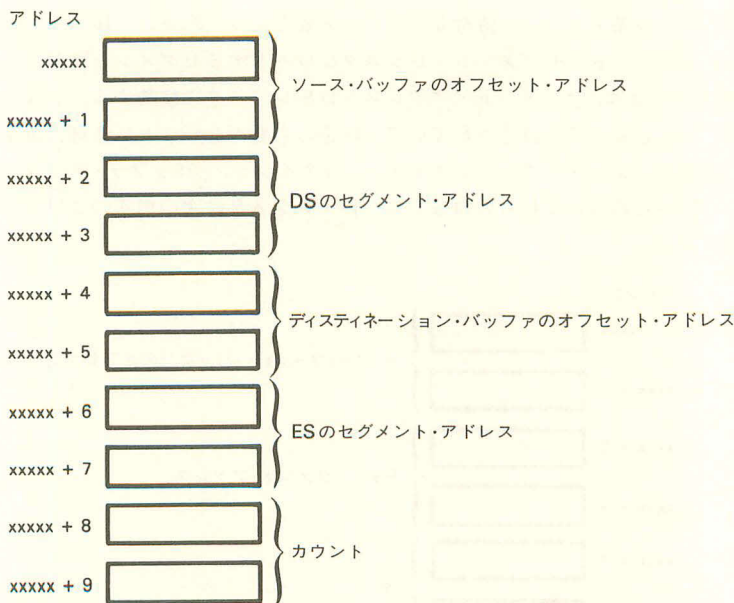
上に示したメモリ・ワードのブロックは，しばしばパラメータ・ブロックと呼ばれる.ブロックの個々のデータの値は，パラメータである.

初期設定のパラメータを得るために，ＤＩレジスタはこのブロックの先頭アドレス（上の例ではxxxx）をロードする.

```
LDS     SI,[DI]
MOV     CX,[DI + 6]
MOV     DI,[DI + 4]
```

図4-3　バッファ移動レジスタ初期設定

もし都合が良ければ，ＤＩレジスタの代わりにＢＸレジスタを用いることができる.

ストリング・プリミティブ命令を用いるときは，ＥＳレジスタにロードされるセグメント・アドレスを含むため，パラメータ・ブロックは以下に示すように拡張される必要がある.

この方法では，1メガバイトのメモリ領域で，任意の位置から別の位置へデータを移動できる．セグメントが指定されていなければ，カレント・セグメント内だけで，データが移動される．

また，このブロックの先頭アドレス（上の例ではxxxx）をＤＩレジスタにロードすると，初期設定は次のようになる．

```
LDS     SI,[DI]
MOV     CX,[DI + 8]
LES     DI,[DI + 4]
```

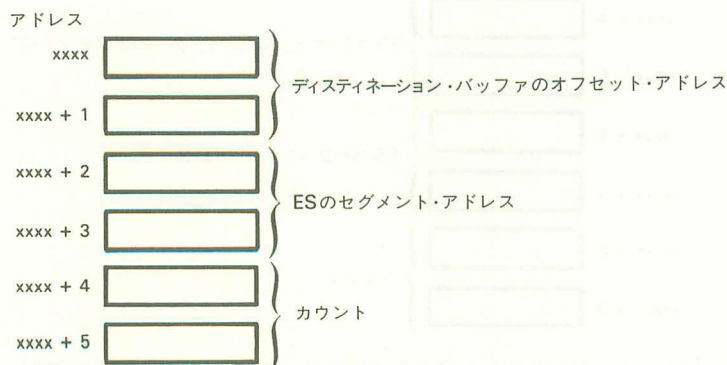
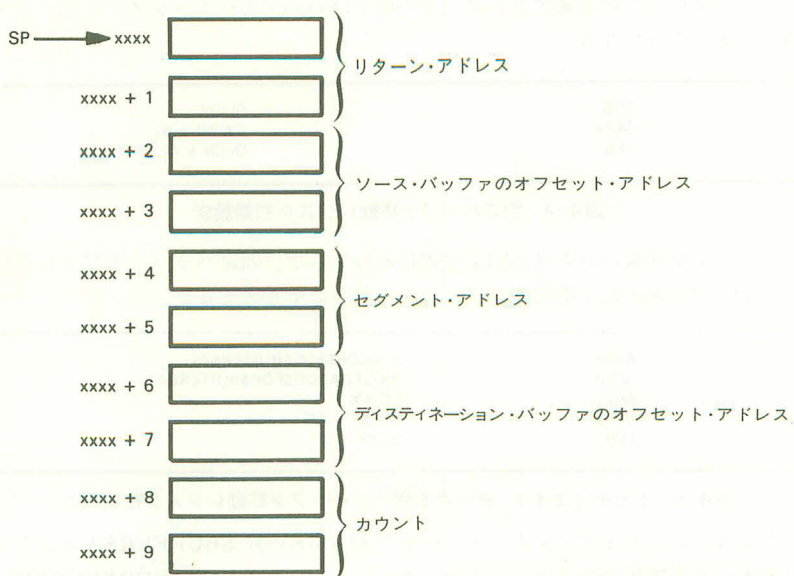
図4-4　別のバッファ移動レジスタ初期設定

1つのバッファが常にメモリの固定位置にあるならば，固定バッファのアドレスはイミディエイト・データとして指定できる．次の一連の命令を考える．

```
MOV     SI,ADDR$FOR$BUFFER$A
MOV     AX,SEGADDR$FOR$BUFFER$A
MOV     DS,AX
MOV     CX,[DI + 4]
LES     DI,[DI]
```

図4-5　イミディエイト・データを用いたバッファ移動レジスタ初期設定

最初の命令では，イミディエイト・データのADDRFORBUFFER$AがＳＩレジスタに移動される．第2の命令は，イミディエイト・データのSEGADDRFORBUFFER$AをＡＸレジスタに移動する．この命令は，8086がイミディエイト・データをセグメン

ト・レジスタに移動する命令を持たないので，必要となる（例外に，16ビットのセグメント・アドレスをコード・セグメント・レジスタにロードするセグメント間ジャンプ命令がある）．第3の命令は，セグメント・アドレスをＤＳレジスタに移動する．しばしばＤＳレジスタはすでに必要な値に設定されていて，修正の必要がないことに注意．第4と第5の命令は，適当なレジスタにカウントとディスティネイション・バッファ・アドレスをロードするために用いられる．これらの命令では，ＤＩレジスタは次の形式のブロックを示している必要がある．

パラメータ（この場合，アドレスとカウントの情報）は，スタックを通してルーチンに受け渡すことができる．バッファからバッファへの移動に対しては,次のように示される．

次の一連の命令は，レジスタの初期設定を行なう．

```
POP     BX      POP RETURN ADDRESS
POP     SI
POP     DS
POP     DI
POP     CX
```

図4-6　スタックとポップ命令によるバッファ移動レジスタ初期設定

この方法では，RET命令を用いてサブルーチンから復帰することは困難となる．しかし，8086ではレジスタをリターン・アドレスとすることができる．したがって，

```
JMP   BX
```

をRETの代わりに用いることができる．この方法が本質的に好ましくないか，あるいはすべてのレジスタが使用されているならば，次の方法が考えられる．

```
PUSH    BP
MOV     BP,SP
MOV     SI, [BP + 4]
MOV     DS,[BP + 6]
MOV     DI, [BP + 8]
MOV     CX,[BP + 10]
```

図4-7　スタックとインダイレクト・アドレッシングによる移動レジスタ初期設定

これらの命令は，必要な初期設定を行なう．そして，ルーチンは次の命令で終了する．

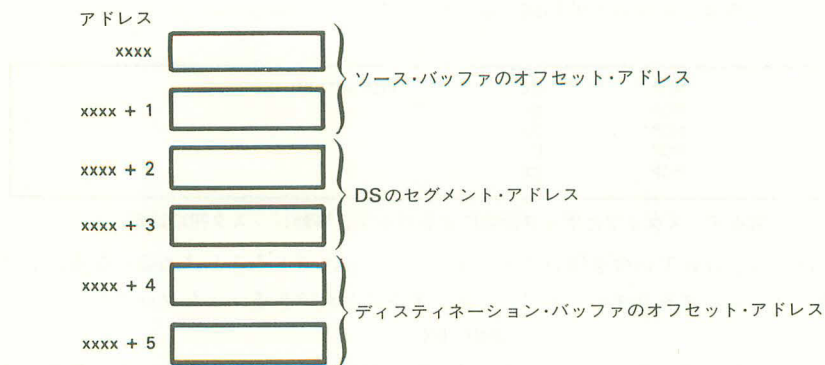
```
MOV     SP, BP
POP     BP
RET     8
```

これは，リターン・アドレスをプログラム・カウンタに移動して，調整されたスタック・ポインタに8を加える．こうして，呼び出し元ルーチンでプッシュされたパラメータを，スタックから取り除く．

もしバッファがカレント・データ・セグメント内に位置すれば，バッファ・アドレスはＬＥＡ(Load Effective Address)命令を用いてロードできる．次の命令は，ＬＥＡ命令を用いてＳＩとＤＩにロードを行ない，ＭＯＶ命令を用いてＣＸにメモリからＣＯＵＮＴのデータをロードしている．

```
LEA     SI,BUFFER$A
LEA     DI,BUFFER$B
MOV     CX,COUNT
```

図4-8　LEA命令を用いたバッファ移動レジスタ初期設定

他の例として，BUFFER$Aの最初の２バイトがバッファ内のバイト数，したがって移動されるバイト数を含むと仮定する．このとき，パラメータ・ブロックは次のようになる．

もし，ＤＩレジスタがこのパラメータ・ブロックを示すとすれば，次の初期設定が用いられる．

```
LDS   SI, [DI]
MOV   DI, [DI+4]
MOV   CX, [SI]
INC   SI
INC   SI
```

２つのバッファ初期設定ルーチンを次に示す．第１のルーチンは８ビットのパターンをバッファ全体に写し，第２のルーチンは16ビットのパターンをバッファ全体に写す．このようなルーチンはバッファをクリヤするのによく用いられ，この場合，８ビットあるいは16ビットの値は０である．バイト長が奇数の短いバッファをクリヤするには最初のルーチンを用い，偶数バイト長のバッファあるいは奇数バイト長の長いバッファ（奇数バイトのクリヤに１バイト処理命令を用いる）には第２のルーチンを用いる．バッファを０でないあるパターンで初期設定する必要が生じる場合がある．たとえば，結果的にＡＳＣＩＩキャラクタのストリングを保持するバッファの初期設定には，ＡＳＣＩＩのスペース・コードが用いられる．

図4-9 と **図4-10** に示すルーチンは，ＤＩレジスタがディスティネーション・バッファを示していることを仮定している．ＡＬあるいはＡＸのレジスタは，バッファ全体に写される８ビットあるいは16ビットの値を含む．ＣＸレジスタは，バッファ内のバイトあるいはワードの数を指定する．

(2) バッファの初期設定

バッファ初期設定ルーチンは，任意のデータをメモリのバッファにロードする．

```
INITIALIZE$LOOP:      MOV    [DI],AL              ;STORE INITIALIZING DATA
                      INC    DI                   ;ADJUST POINTER
                      DEC    CX                   ;DECREMENT AND BRANCH
                      JNZ    INITIALIZE$LOOP      ;If not done
                      RET
```

図4-9 バッファ初期設定（8ビット・データ要素）

```
INITIALIZE$LOOP:      MOV    [DI],AX              ;STORE INITIALIZING DATA
                      INC    DI
                      INC    DI
                      DEC    CX
                      JNZ    INITIALIZE$LOOP
                      RET
```

図4-10 バッファ初期設定（16ビット・データ要素）

2つのバッファ初期設定プログラムで，ＢＸあるいはＳＩのレジスタをＤＩの代わりに用いることができる．

しばしば，バッファの最初のnバイトはバッファの記述に用いられる．たとえば，バッファ全体の長さと先頭の空のバイトに対するディスプレイスメントは，バッファの最初の2バイトにストアされる．このバッファ記述用のバイトは，データがバッファに書かれるときは，調整される必要がある．

バッファ初期設定ルーチンは，バッファからバッファへの移動ルーチンで述べたと同じく，それ自体レジスタを初期設定する必要がある．

一般に，アドレスやカウントの情報は，以下のどれかの方法でルーチンに渡される．

- パラメータ・ブロック
- スタック
- イミディエイト・データ
- ＬＥＡ命令で用いられるアドレス

(3) バッファの変換

バッファが変換されるとき，バッファ内のすべての要素は変換用のテーブルを用いて変換される．変換テーブルは，要素が持つすべての初期値に対して，直接に置換する値を与える．たとえば，バッファが1バイト要素から成るならば，各要素が持つことのできる，256個の可能性のある初期値が存在し，同様に同じ要素を持つことのできる256個の変換値が存在する．変換テーブルは，各初期値と変換値との結合を行なう．おそらく最も多く見られる変換テーブルは，ＡＳＣＩＩとＥＢＣＤＩＣのキャラクタ間の変換で，各々はバイト値として符号化されている．この場合，ＡＳＣＩＩキャラクタのバッファが変換されれば，結果は等価なＥＢＣＤＩＣキャラクタのバッファになる．

バッファが変換される次の2つの方法を考える．

1．バッファ内のデータが変換されてそのバッファに残る．

2．データが変換されて1つのバッファから他へ移動する．

図4-11のルーチンは，データを移動しないで変換する．このルーチンは，ＢＸレジスタ

が変換テーブルのアドレスを含み，ＳＩレジスタが変換されるべきバッファのアドレスを含み，ＣＸレジスタが変換されるべきデータ要素の数を含むと仮定している．

```
TRANSLATE$LOOP:    MOV     AL,[SI]               ;LOAD FROM BUFFER
                   XLAT                          ;INDEX INTO TABLE
                   MOV     [SI],AL               ;STORE CONVERTED DATA INTO BUFFER
                   INC     SI                    ;POINT AT NEXT ELEMENT
                   DEC     CX                    ;DECREMENT AND TEST FOR DONE
                   JNZ     TRANSLATE$LOOP
                   RET
```

図4-11 バッファ内容の変換

図4-11のルーチンは，変換される要素が256バイト・テーブルに写されることを仮定している．この仮定で，ＸＬＡＴ命令を用いることが可能となる．変換されるべきデータ要素が16ビットのデータ単位であれば，より大きいテーブルが必要となる．図4-12のルーチンは，16ビットのデータ要素を65キロバイト* のテーブルに写し，8ビットの結果を得る．

```
TRANSLATE$LOOP:        MOV     DI,[SI]              ;LOAD ELEMENT
                       MOV     AX,[BX + DI]         ;USE ELEMENT AS INDEX
                       MOV     [SI], AX             ;STORE RESULT
                       INC     SI                   ;UPDATE POINTERS
                       INC     SI
                       DEC     CX                   ;DECREMENT AND TEST
                       JNZ     TRANSLATE$LOOP       ;FOR DONE
                       RET
```

図4-12 16ビット・データ要素の変換

```
TRANSLATE$LOOP: MOV     AL,[SI]        ;LOAD ELEMENT FROM SOURCE BUFFER
                XLAT                   ;TRANSLATE DATA
                MOV     [DI],AL        ;STORE CONVERTED DATA IN DESTINATION BUFFER
                INC     SI             ;UPDATE POINTERS
                INC     DI
                DEC     CX             ;DECREMENT AND TEST FOR DONE
                JNZ     TRANSLATE$LOOP
                RET
```

図4-13 バッファからバッファへの変換移動

図4-13のルーチンは，データを変換して，１つのバッファから他へ移動する．このルーチンは，ＢＸレジスタが変換テーブルのアドレスを含み，ＳＩレジスタが変換されるバッファのアドレスを含み，ＤＩレジスタが変換されたデータのストアされるバッファのアドレスを含み，ＣＸレジスタが変換されるデータ要素の数を含むと仮定している．

図4-13のルーチンは，両方のバッファがＤＳレジスタで示されるセグメント内に存在す

＊ $2^{16}=65,536$（訳者注）

ることを仮定している．

多くの変換ルーチンでは，変換バッファ内のすべての要素が特定の境界値内にあるかのチェックも行なう．図4-11から図4-13までのルーチンは，そのようなチェックを含むように，この章の後で更新を行なう．

前記のルーチンのレジスタ初期設定は，バッファからバッファへの移動ルーチンに用いられている初期設定の方法と類似している．

4.1.2 ＣＰＵの状態の退避

8086は14個の16ビット・レジスタを持つ．ほとんどの場合，多くのレジスタを持つことは非常に望ましい．しかし，適当な処理が行なわれる間，ＣＰＵ全体の状態がセーブされる必要があり，次いでＣＰＵの状態が復元される必要のあるときに，多数のレジスタはあまり価値がない．

たとえば，ハードウエアあるいはソフトウエアの割り込みが発生すると，インタラプト・アクノリッジ・シーケンスで，8086は，フラグ・レジスタ，プログラム・カウンタ，コード・セグメント・レジスタの内容をセーブする．インタラプト・サービス・ルーチンでは，ＣＰＵの完全な状態をセーブしなければならない．次に示す一連の命令は，この作業を行なう．

レジスタをプッシュすべき特定の順序は存在しない．ただし，レジスタはそれがプッシュされた逆の順序で復元される必要がある．もしレジスタが図4-14に示す方法でセーブされれば，次に示されている方法がレジスタを復元するために用いられなければならない．

```
PUSH ES
PUSH DS
PUSH SI
PUSH DI
PUSH BP
PUSH DX
PUSH CX
PUSH BX
PUSH AX
```

図4-14　8086レジスタのセーブ

```
POP AX
POP BX
POP CX
POP DX
POP BP
POP DI
POP SI
POP DS
POP ES
```

図4-15　8086レジスタの回復

ＣＰＵ全体の状態をセーブするためには，11バイトの命令コードと 110 クロックの期間を要する．この 110 サイクルには，8086が割り込みに応答するための時間は含まれていない．ハードウエア・インタラプトに対して，8086は割り込みを受け付けるために62クロックの期間を要する．ソフトウエア・インタラプトに対して，8086は割り込みを受け付けるために51から53クロックの期間を要する．この応答のためのサイクルは，割り込みが発生した間の命令実行に続いている．

したがって，ＣＰＵ全体の状態の退避は 172 クロックの期間をも要し，これは 5 MHz の8086では32.4マイクロ秒になる．ＣＵＰの状態の復元には 110 クロックの期間を要し，さらにＩＲＥＴ（Interrupt Return）命令の24クロックが加わる．以上から8086のＣＰＵの状態を退避して復元するためには，１つの割り込みについて，306 クロックの期間あるいは61.2マイクロ秒を要する．

4.1.3 セグメント・レジスタの初期設定

プログラムがオペレーティング・システムと共に動作するように書かれていれば，オペレーティング・システムは一般にセグメント・レジスタの初期設定を行ない，続いて必要に応じてその内容を変更する．プログラムがオペレーティング・システムの恩恵を受けずに動作するように書かれていれば，セグメント・レジスタの初期設定を行なう必要がある．**図4-16**に示す命令は，セグメント・レジスタの初期設定を行なう．

```
MOV     AX, IMM$DATA$FOR$DS         ;LOAD IMMEDIATE DATA INTO AX
MOV     DS, AX
MOV     AX, IMM$DATA$FOR$ES         ;LOAD IMMEDIATE DATA INTO AX
MOV     ES, AX
MOV     AX, IMM$DATA$FOR$SS         ;LOAD IMMEDIATE DATA INTO AX
MOV     SS, AX
```

図4-16　イミディエイト・データによるESレジスタの初期設定

セグメント・レジスタの初期設定を行なうもう１つの方法は，**図4-17**に示すように，メモリからセグメント・レジスタにデータを直接移動すればよい．

```
MOV          DS, CS: DATA$FOR$DS
MOV          ES, CS: DATA$FOR$ES
MOV          SS, CS: DATA$FOR$SS
```

図4-17　コード・セグメント指定によるESレジスタの初期設定

セグメント・レジスタのＥＳとＳＳに対するデータがＤＳで示されるセグメントに含まれていれば，第２と第３の命令のセグメント・プレフィックスは省略できる．

8086は，特定の初期設定の命令について，特殊な保護を行なう．ＳＳとＳＰのレジスタが連続したＭＯＶ命令で初期設定されるとき，8086はこのＭＯＶ命令の間で割り込みが発生するのを禁止する．したがって，

```
MOV     SS, CS:DATA$FOR$SS
MOV     SP, DATA$FOR$SP
```

の一連の命令では，割り込みは生じない．

4.2　算術演算命令

8086の算術演算命令には，次の5つのタイプがある．

1．加算命令
2．減算命令
3．乗算命令
4．除算命令
5．比較命令

上記分類の各々は，比較命令を除いて，ASCII/BCDの操作に用いることができる．

4.2.1 加算命令

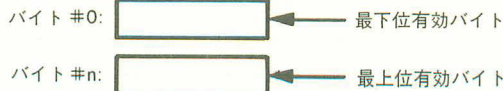
種々のタイプの加算を行なう命令を**表4-2**に示す．**図4-18**，**4-19**，**4-20**は，いろいろな加算命令の使用を示している．各ルーチンは，加算される数あるいはストリングはデータ・セグメントに存在し，次の順序になっていると仮定している．

(1)　1組の複数ワード数値の和

図4-18のルーチンは，ＳＩとＤＩのレジスタが加算される複数ワード数値の先頭アドレスを含み，ＣＸレジスタが加算されるワード数を含むと仮定している．結果は，ＤＩレジスタで示されるストリングにストアされる．

```
START:              CLC                          ;CLEAR CARRY FOR INITIAL ADDITION
ADDITION$LOOP:      MOV     AX,[SI]              ;LOAD FROM INITIAL STRING
                    ADC     [DI],AX              ;ADD AX TO MEMORY
                    INC     SI                   ;UPDATE POINTERS
                    INC     SI
                    INC     DI
                    INC     DI
                    DEC     CX
                    JNZ     ADDITION$LOOP
                    RET
```

図4-18　複数ワード加算

ストリング・プリミティブとＬＯＯＰ命令によって，メモリの数とこのルーチン実行に必要な時間が減少できる．

(2)　1組の複数バイトＢＣＤ数値の和

図4-19のルーチンは，ＳＩとＤＩのレジスタが加算されるＢＣＤストリングの先頭アドレスを含み，ＣＸレジスタが各ストリングのＢＣＤバイト数を含むと仮定している．結果はＤＩレジスタで示されるストリングにストアされる．

表4－2　8086の加算命令

ニーモニック	オペランド	オブジェクト・コード	バイト	クロック	ステータス O	D	I	T	S	Z	A	P	C	動作内容
ADC	mem/reg_1, mem/reg_2	000100dw mod rrr r/m (DISP) (DISP)	2，3または4	reg - reg: 3 mem - reg: 9 + EA reg - mem: 16 + EA	X				X	X	X	X	X	〔mem/reg_1〕←〔mem/reg_1〕＋〔mem/reg_2〕＋〔CF〕 mem/reg_2で示されるメモリ位置あるいはレジスタの8あるいは16ビットの内容とキャリー・フラグを，mem/reg_1で示されるメモリ位置あるいはレジスタの8あるいは16ビットの内容に加算する．
ADC	mem/reg, data	100000sw mod 010 r/m (DISP) (DISP) kk jj（sw＝01のとき）	3，4，5または6	reg: 4 mem: 17 + EA	X				X	X	X	X	X	〔mem/reg〕←〔mem/reg〕＋data＋〔CF〕 8あるいは16ビットのイミディエイト・データとキャリー・フラグを，mem/regで示されるメモリ位置あるいはレジスタの8あるいは16ビットの内容に加算する．
ADC	ac, data	0001010w kk jj（w＝1のとき）	2または3	4	X				X	X	X	X	X	〔ac〕←〔ac〕＋data＋〔CF〕 8あるいは16ビットのイミディエイト・データとキャリー・フラグを，AL（8ビット操作）あるいはAX（16ビット操作）のレジスタに加算する．
ADD	mem/reg_1, mem/reg_2	000000dw mod rrr r/m (DISP) (DISP)	2，3または4	reg - reg: 3 mem - reg: 9 + EA reg - mem: 16 + EA	X				X	X	X	X	X	〔mem/reg_1〕←〔mem/reg_1〕＋〔mem/reg_2〕 mem/reg_2で示されるメモリ位置あるいはレジスタの8あるいは16ビットの内容を，mem/reg_1で示されるメモリ位置あるいはレジスタの8あるいは16ビットの内容に加算する．
ADD	mem/reg, data	100000sw mod 000 r/m (DISP) (DISP) kk jj（sw＝01のとき）	3，4，5または6	reg: 4 mem: 17 + EA	X				X	X	X	X	X	〔mem/reg〕←〔mem/reg〕＋data 8あるいは16ビットのイミディエイト・データを，mem/regで示されるメモリ位置あるいはレジスタの8あるいは16ビットの内容に加算する．
ADD	ac,data	0000010w kk jj（w＝1のとき）	2または3	4	X				X	X	X	X	X	〔ac〕←〔ac〕＋data 8あるいは16ビットのイミディエイト・データを，AL（8ビット操作）あるいはAX（16ビット操作）のレジスタに加算する．
INC	mem/reg	1111111w mod 000 r/m (DISP) (DISP)	2，3または4	reg: 3 mem: 15 + EA	X				X	X	X	X		〔mem/reg〕←〔mem/reg〕＋1 mem/regで示されるメモリ位置あるいはレジスタの8あるいは16ビットの内容を，1だけ増やす．

表4-2　8086の加算命令(続き)

ニーモニック	オペランド	オブジェクト・コード	バイト	クロック	ステータス									動作内容
					O	D	I	T	S	Z	A	P	C	
INC	reg	01000rrr	1	2	X				X	X	X	X		〔reg〕←〔reg〕＋1 指定されたレジスタの16ビットの内容を，1だけ増やす．
AAA		37	1	4	?				?	?	X	?	X	ASCIIによる加算後にALレジスタの内容をアジャストする．
DAA		27	1	4	?				X	X	X	X	X	10進による加算後にALレジスタの内容をアジャストする．

```
START:                CLC                              ;CLEAR CARRY FOR INITIAL ADDITION
BCD$ADDITION$LOOP:    MOV      AL,[SI]                 ;LOAD FROM STRING A
                      ADC      AL,[DI]                 ;ADD FROM STRING B
                      DAA                              ;PERFORM BCD ADJUST
                      MOV      [DI],AL                 ;STORE RESULT
                      INC      SI                      ;UPDATE POINTERS
                      INC      DI
                      DEC      CX                      ;DECREMENT AND TEST
                      JNZ      BCD$ADDITION$LOOP       ;FOR DONE
                      RET
```

図4-19　複数バイトのBCD加算

(3)　1組の複数バイトＡＳＣＩＩストリングの和

図4-20のルーチンは，ＳＩとＤＩのレジスタが加算される２つのＡＳＣＩＩストリングの先頭アドレスを含むと仮定している．ＣＸレジスタは，各ストリングのＡＳＣＩＩバイト数を含む．結果は，ＤＩレジスタで示されるストリングにストアされる．

```
                      CLC
ASCII$ADDITION$LOOP:  MOV      AL,[SI]                 ;LOAD FROM STRING A
                      ADC      AL,[DI]                 ;ADD STRING B
                      AAA                              ;PERFORM AN ADJUST
                      MOV      [DI],AL                 ;STORE RESULT
                      INC      SI                      ;ADJUST POINTERS
                      INC      DI
                      DEC      CX                      ;DECREMENT AND TEST
                      JNZ      ASCII$ADDITION$LOOP     ;FOR DONE
                      RET
```

図4-20　複数バイトのASCII加算

図4-19と図4-20のルーチンは共に，操作を行なうのに必要なバイト数と時間を減らすために，ストリング・プリミティブを用いることができる．

前記の加算ルーチンに対して，加算される数が次の形式である場合を考える．

バイト＃０　[　　　]　オペランドの最上位バイト

バイト＃n　[　　　]　オペランドの最下位バイト

この場合，加算ルーチンは図4-18から図4-20のものとは，次の２つの主要な点で異なる．

1．初期設定が異なる．初期設定では，レジスタは複数バイト数値の最初ではなく最後を示している．

2．ポインタは，増加ではなく，減少する．

この差を補償するためには，変更された先頭アドレスを適当なアドレス・レジスタにロードする必要がある．続いて，アドレスは減少させなければならない．

4.2.2 減算命令

減算命令を表４-3に示す．前節の複数バイトの加算ルーチンを減算に直したものは容易に得られる．このルーチンの作成は，読者に練習問題として残す．

表4-3　8086の減算命令

ニーモニック	オペランド	オブジェクト・コード	バイト	クロック	ステータス									動作内容
					O	D	I	T	S	Z	A	P	C	
SUB	mem/reg$_1$, mem/reg$_2$	001010dw mod rrr r/m (DISP) (DISP)	2, 3または4	reg - reg: 3 mem - reg: 9 + EA reg - mem: 16 + EA	X				X	X	X	X	X	〔mem/reg$_1$〕←〔mem/reg$_1$〕−〔mem/reg$_2$〕 mem/reg$_1$で示されるメモリ位置あるいはレジスタの8あるいは16ビットの内容から,mem/reg$_2$で示されるメモリ位置あるいはレジスタの8あるいは16ビットの内容を減じる.
SUB	mem/reg, data	100000sw mod 101 r/m (DISP) (DISP) kk jj (sw=01のとき)	3, 4, 5または6	reg: 4 mem: 17 + EA	X				X	X	X	X	X	〔mem/reg〕←〔mem/reg〕−data mem/regで示されるメモリ位置あるいはレジスタの8あるいは16ビットの内容から，8あるいは16ビットのイミディエイト・データを減じる.
SUB	ac,data	0010110w kk jj (w=1のとき)	2または3	4	X				X	X	X	X	X	〔ac〕←〔ac〕−data AL（8ビット操作）あるいはAX（16ビット操作）のレジスタから，8あるいは16ビットのイミディエイト・データを減じる.
SBB	mem/reg$_1$, mem/reg$_2$	000110dw mod rrr r/m (DISP) (DISP)	2, 3または4	reg - reg: 3 mem - reg: 9 + EA reg - mem: 16 + EA	X				X	X	X	X	X	〔mem/reg$_1$〕←〔mem/reg$_1$〕−〔mem/reg$_2$〕−〔CF〕 mem/reg$_1$で示されるメモリ位置あるいはレジスタの8あるいは16ビットの内容から，mem/reg$_2$で示されるメモリ位置あるいはレジスタの8あるいは16ビットの内容とキャリー・フラグを減じる.
SBB	mem/reg, data	100000sw mod 011 r/m (DISP) (DISP) kk jj (sw=01のとき)	3, 4, 5または6	reg: 4 mem: 17 + EA	X				X	X	X	X	X	〔mem/reg〕←〔mem/reg〕−data−〔CF〕 mem/regで示されるメモリ位置あるいはレジスタの8あるいは16ビットの内容から，8あるいは16ビットのイミディエイト・データとキャリー・フラグを減じる.
SBB	ac,data	0001110w kk jj (w=1のとき)	2または3	4	X				X	X	X	X	X	〔ac〕←〔ac〕−data−〔CF〕 AL（8ビット操作）あるいはAX（16ビット操作）のレジスタから，8あるいは16ビットのイミディエイト・データとキャリー・フラグを減じる.
DEC	mem/reg	1111111w mod 001 r/m (DISP) (DISP)	2, 3または4	reg: 3 mem: 15 + EA	X				X	X	X	X		〔mem/reg〕←〔mem/reg〕−1 mem/regで示されるメモリ位置あるいはレジスタの8あるいは16ビットの内容を1だけ減じる.

表 4-3　8086の減算命令（続き）

ニーモニック	オペランド	オブジェクト・コード	バイト	クロック	ステータス									動　作　内　容
					O	D	I	T	S	Z	A	P	C	
DEC	reg	01001rrr	1	2	X				X	X	X	X		〔reg〕←〔reg〕－1 指定されたレジスタの16ビットの内容を1だけ減じる．
AAS		3F	1	4	?				?	?	X	?	X	ASCIIによる減算後のALレジスタの内容をアジャストする．
DAS		2F	1	4	?				X	X	X	X	X	10進による減算後のALレジスタの内容をアジャストする．
NEG	mem/reg	1111011w mod 011 r/m (DISP) (DISP)	2, 3または4	reg: 3 mem: 16＋EA	X				X	X	X	X	X	〔reg〕←$\overline{〔reg〕}$＋1 mem/regで示されるメモリ位置あるいはレジスタの，8あるいは16ビットの内容の2の補数をとる．

4.2.3 乗算命令

種々のタイプの乗算を行なう8086の命令を**表4-4**に示す.

図4-21と**図4-22**のルーチンは, 8086の乗算命令の代表的な使用を示している.

(1) 32ビットと32ビットの乗算

図4-21のルーチンは, 2つの32ビット符号なし数値を乗じて, 64ビットの結果を与える. このルーチンは次の形式のデータ・ブロックに作用する.

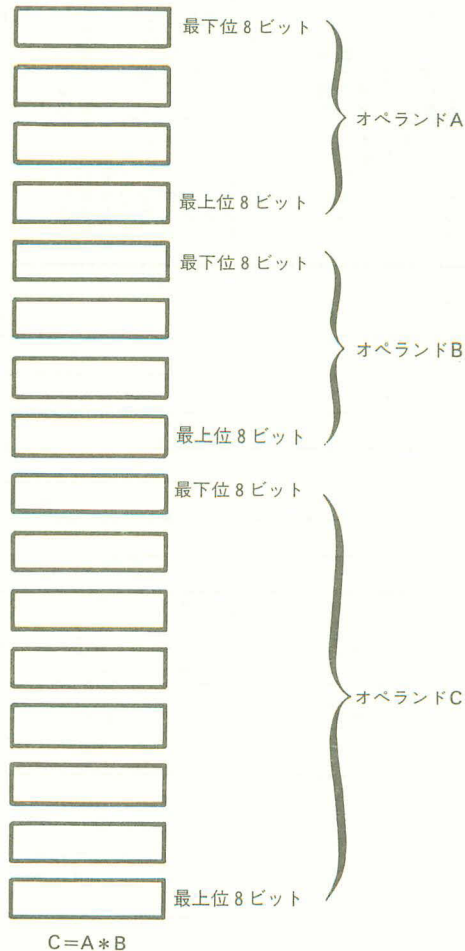



図4-21のルーチンは, BXレジスタがこのブロックを示していることを仮定している.

このような処理に対するもう1つの方法は, 8086システムへの8087ニューメリック・コープロセッサの付加である. 8087は, 単精度と倍精度の浮動小数点演算と同様に, 32ビットと64ビットの整数演算を行なう.

表4-4　8086の乗算命令

ニーモニック	オペランド	オブジェクト・コード	バイト	クロック	ステータス									動作内容
					O	D	I	T	S	Z	A	P	C	
MUL	reg (8-bit)	11110110 11100 reg	2	70 → 77	X				?	?	?	?	X	w＝0のとき,〔AX〕←〔AL〕・〔mem/reg〕 w＝1のとき,〔DX〕:〔AX〕←〔AX〕・〔mem/reg〕 mem/regで示されるメモリ位置あるいはレジスタの8あるいは16ビットの内容と, AL（8ビット操作）あるいはAX（16ビット操作）のレジスタの内容を乗じる.結果は, 8×8ビット操作の場合, AXレジスタにストアされ, 16×16ビット操作の場合, DXレジスタ（上位16ビット）とAXレジスタ（下位16ビット）にストアされる. 内容は符号なしの乗算である. 実行時間は, 8ビット操作で7クロック, 16ビット操作で15クロックの変動がある.
	reg (16-bit)	11110111 11100 reg	2	118 → 133										
	mem (8-bit)	11110110 mod 100 r m (DISP) (DISP)	2, 3または4	(76 → 83) + EA										
	mem (16-bit)	11110111 mod 100 r m (DISP) (DISP)	2, 3または4	(124 → 139) + EA										
IMUL	reg (8-bit)	11110110 11101 reg	2	80 → 98	X				?	?	?	?	X	w＝0のとき,〔AX〕←〔AL〕・〔mem/reg〕 w＝1のとき,〔DX〕:〔AX〕←〔AX〕・〔mem/reg〕 mem/regで示されるメモリ位置あるいはレジスタの8あるいは16ビットの内容と, AL（8ビット操作）あるいはAX（16ビット操作）のレジスタの内容を乗じる.結果は, 8×8ビット操作の場合, AXレジスタにストアされ, 16×16ビット操作の場合, DXレジスタ（上位16ビット）とAXレジスタ（下位16ビット）にストアされる. 内容は符号付き乗算である. 実行時間は, 8ビット操作で18クロック, 16ビット操作で26クロックの変動がある.
	reg (16-bit)	11110111 11101 reg	2	128 → 154										
	mem (8-bit)	11110110 mod 101 r m (DISP) (DISP)	2, 3または4	(86 → 104) + EA										
	mem (16-bit)	11110111 mod 101 r m (DISP) (DISP)	2, 3または4	(134 → 160) + EA										
AAM		D4 0A	2	83	?				X	X	?	X	?	2つのアンパック形式の10進オペランドの乗算後に, アンパック形式の10進の結果を得るためにAXの積をアジャストする.

```
                    MOV     AX, [BX]            ;MULTIPLY LOW-ORDER 16 BITS
                    MUL     [BX + 4]            ;BY LOW-ORDER 16 BITS

                    MOV     [BX + 8],AX         ;SAVE RESULT, WHICH IS IN AX
                    MOV     [BX + 10],DX        ;AND DX

                    MOV     AX, [BX]            ;MULTIPLY LOW-ORDER 16 BITS OF
                                                ;OPERAND A BY HIGH-ORDER 16 BITS
                    MUL     [BX + 6]            ;OF OPERAND B

                    ADD     [BX + 10],AX        ;ADD TO PREVIOUS RESULT
                    ADC     [BX + 12],DX        ;ASSUME RESULT BYTES
                    JNC     NEXT$MUL            ;ARE INITIALLY ZERO
                    INC     [BX + 14]
NEXT$MUL:           MOV     AX,[BX + 2]         ;MULTIPLY HIGH-ORDER 16 BITS OF
                                                ;OPERAND A BY LOW-ORDER 16 BITS
                    MUL     [BX + 4]            ;OF OPERAND B

                    ADD     [BX + 10],AX        ;ADD TO PREVIOUS RESULT
                    ADC     [BX + 12],DX
                    JNC     HIGH$ORDER$MUL
                    INC     [BX + 14]           ;SAVE CARRY
HIGH$ORDER$MUL:     MOV     AX, [BX + 2]        ;MULTIPLY HIGH-ORDER 16 BITS
                                                ;OF OPERAND A BY HIGH-ORDER 16
                    MUL     [BX + 6]            ;BITS OF OPERAND B

                    ADD     [BX + 12],AX        ;ADD TO PREVIOUS RESULT
                    ADC     [BX + 14],DX        ;ADD TO PREVIOUS RESULT
                    RET
```

図4-21　32ビットと32ビットの乗算

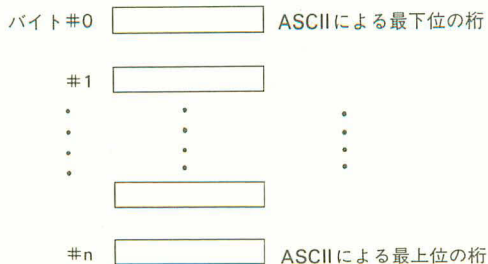
(2)　ＡＳＣＩＩによる乗算

図4-22は，ＡＳＣＩＩストリングとＡＳＣＩＩの１桁の乗算を行なう．結果は，アンパック形式ＢＣＤ桁のストリングである．ルーチンは，ＡＳＣＩＩストリングが次のように構成されていることを仮定している．

バイト#0　[　　]　ASCIIによる最下位の桁

#1　[　　]

・　　・　　・

[　　]

#n　[　　]　ASCIIによる最上位の桁

ルーチンではさらに，ＳＩレジスタがＡＳＣＩＩストリングを示し，ＤＬレジスタが乗数であるＡＳＣＩＩの１桁を含み，ＤＩレジスタが結果のＢＣＤストリングがストアされるメモリ位置を示し，ＣＸレジスタが被乗数の桁数を含むと仮定している．ＢＣＤストリングでストアされる結果は，次の形式となる．

バイト#0 [ ] BCDによる最下位の桁

#1 [ ]

・ ・ ・

[ ]

#n+1 [ ] BCDによる最上位の桁

```
                     MOV    [DI], 0                ;CLEAR INITIAL BYTE OF BCD STRING
                     AND    DL,0FH                 ;AND OFF BITS 4 AND 5 OF MULTIPLIER
MULTIPLY$NEXT$BYTE:  MOV    AL,[SI]                ;LOAD MULTIPLICAND
                     INC    SI
                     AND    AL,0FH                 ;CLEAR UPPER NIBBLE
                     MUL    DL                     ;MULTIPLY BCD * BCD
                     AAM                           ;ADJUST RESULT
                     ADD    AL,[DI]                ;ADD IN BCD
                     AAA
                     MOV    [DI],AL                ;STORE RESULT
                     INC    DI
                     MOV    [DI],AH
                     DEC    CX                     ;DECREMENT AND TEST FOR DONE
                     JNZ    MULTIPLY$NEXT$BYTE
                     RET
```

図4-22　ASCIIによる乗算

4.2.4 除算命令

種々の除算処理を行なう8086の命令を**表4-5**に示す.

図4-23のルーチンは，8086の除算命令の使用を示している.

(1) ＡＳＣＩＩによる除算

図4-23のルーチンは，ＡＳＣＩＩストリングのＡＳＣＩＩ１桁による除算を行なう．結果はＢＣＤ桁のストリングである．ルーチンは，ＡＳＣＩＩストリングが次のように構成されていることを仮定している．

表4-5　8086の除算命令

ニーモニック	オペランド	オブジェクト・コード	バイト	クロック	ステータス O	D	I	T	S	Z	A	P	C	動作内容
DIV	reg (8-bit)	11110110 11110 reg	2	80 → 90	?				?	?	?	?	?	w＝0のとき，〔AH〕余り，〔AL〕商 ←〔AX〕/〔mem/reg〕 w＝1のとき，〔DX〕余り，〔AX〕商 ←〔DX〕:〔AX〕/〔mem/reg〕 16ビット操作の場合はAXレジスタを，あるいは32ビット操作の場合はDXレジスタ（上位16ビット）とAXレジスタ（下位16ビット）を，mem/regで示されるメモリ位置あるいはレジスタの8あるいは16ビットの内容で割る．16ビットを8ビットで割る場合，商はALに置かれ，余りはAHにストアされる．32ビットを16ビットで割る場合，商はAXレジスタに置かれ，余りはDXレジスタに置かれる．これは符号なし除算操作である． 実行時間は，8ビット操作では10クロックの，16ビット操作では18クロックの変動がある．この変動はデータに依存する．
	reg (16-bit)	11110111 11110 reg	2	144 → 162										
	mem (8-bit)	11110110 mod 110 r/m (DISP) (DISP)	2，3または4	(86 → 96) ＋EA										
	mem (16-bit)	11110111 mod 110 r/m (DISP) (DISP)	2，3または4	(150 → 168) ＋EA										
IDIV	reg (8-bit)	11110110 11111 reg	2	101 → 112	?				?	?	?	?	?	w＝0のとき，〔AH〕余り，〔AL〕商 ←〔AX〕/〔mem/reg〕 w＝1のとき，〔DX〕余り，〔AX〕商 ←〔DX〕:〔AX〕/〔mem/reg〕 16ビット操作の場合はAXレジスタを，あるいは32ビット操作の場合はDXレジスタ（上位16ビット）とAXレジスタ（下位16ビット）を，mem/regで示されるメモリ位置あるいはレジスタの8あるいは16ビットの内容で割る．16ビットを8ビットで割る場合，商はALに置かれ，余りはAHにストアされる．32ビットを16ビットで割る場合，商はAXレジスタに置かれ，余りはDXレジスタに置かれる．これは符号付き除算操作である． 実行時間は，8ビット操作では11クロックの，16ビット操作では19クロックの変動がある．この変動はデータに依存する．
	reg (16-bit)	11110111 11111 reg	2	165 → 184										
	mem (8-bit)	11110110 mod 111 r/m (DISP) (DISP)	2，3または4	(107 → 118) ＋EA										
	mem (16-bit)	11110111 mod 111 r/m (DISP) (DISP)	2，3または4	(171 → 190) ＋EA										

表 4 -5　8086の除算命令(続き)

ニーモニック	オペランド	オブジェクト・コード	バイト	クロック	ステータス									動作内容
					O	D	I	T	S	Z	A	P	C	
CBW		98	1	2										〔AH〕←〔AL7〕 ALレジスタの符号ビット，ビット7をAHレジスタに拡張する．
CWD		99	1	5										〔DX〕←〔AX15〕 AXレジスタの符号ビット，ビット15をDXレジスタに拡張する．
AAD		D5 0A	2	60	?				X	X	?	X	?	アンパック形式の10進の商を得るために，アンパック形式の10進の除数の除算に先立って，ALの被除数の10進アジャストを行なう．

ＳＩレジスタはＡＳＣＩＩストリングを示し，ＤＬレジスタは除数であるＡＳＣＩＩの1桁を含み，ＤＩレジスタは結果のＢＣＤストリングがストアされるメモリ位置を示し，ＣＸレジスタは被除数の桁数を含む．ＢＣＤストリングでストアされる結果は，次の形式となる．

```
                    AND     DL,0FH              ;CLEAR HIGH-ORDER NIBBLE
                    XOR     AH,AH               ;CLEAR AH
DIVIDE$NEXT$BYTE:   MOV     AL,[SI]             ;LOAD BYTE FROM ASCII STRING
                    INC     SI
                    AND     AL,0FH              ;CLEAR BITS 4 AND 5
                    AAD                         ;ADJUST USING AH
                    DIV     DL
                    MOV     [DI],AL             ;STORE RESULT
                    INC     DI
                    DEC     CX                  ;DECREMENT AND TEST FOR DONE
                    JNZ     DIVIDE$NEXT$BYTE
                    RET
```

図4-23　ASCIIによる除算

(2) 64ビットの除算

64ビットの被除数を32ビットの除数で割ることは，8086では容易なことではない．ＤＩＶとＩＤＩＶの命令は，この機能を行なうときに特に有用ではない．64ビット数値を32ビットで割るためには，減算とシフトのアルゴリズムを用いる必要がある．このようなルーチンの作成は，重要な作業で，ここでの解説の範囲を越えている．

このような処理に対するもう１つの方法は，8086システムへの8087ニューメリック・コープロセッサの付加である．8087は，単精度と倍精度の浮動小数点演算と同様に，32ビットと64ビットの整数演算を行なう．

4.2.5 比較命令

8086の比較命令を**表４-６**に示す．これらは減算命令のように動作するが，レジスタあるいはメモリ位置に結果を戻さない．減算操作はステータス・フラグを設定するためにだけ用いられる．

比較命令の使用を**図4-24**から**図4-27**に示す．

表4-6　8086の比較命令

ニーモニック	オペランド	オブジェクト・コード	バイト	クロック	ステータス									動作内容
					O	D	I	T	S	Z	A	P	C	
CMP	mem/reg_1, mem/reg_2	001110dw mod rrr r/m (DISP) (DISP)	2, 3または4	reg-reg: 3 mem-reg: 9＋EA reg-mem: 9＋EA	X				X	X	X	X	X	〔mem/reg_1〕－〔mem/reg_2〕 mem/reg_1で示されるメモリ位置あるいはレジスタの8あるいは16ビットの内容から，mem/reg_2で示されるメモリ位置あるいはレジスタの8あるいは16ビットの内容の減算を行なう．結果はフラグを設定するために用いられてから捨てられる．
CMP	mem/reg, data	100000sw mod 111 r/m (DISP) (DISP) kk jj (sw＝01のとき)	3, 4, 5または6	reg: 4 mem: 10＋EA	X				X	X	X	X	X	〔mem/reg〕－data mem/regで示されるメモリ位置あるいはレジスタの8あるいは16ビットの内容から，8あるいは16ビットのイミディエイト・データの減算を行なう．結果はフラグを設定するために用いられてから捨てられる．
CMP	ac,data	0011110w kk jj (w＝1のとき)	2または3	4	X				X	X	X	X	X	〔ac〕－data AL（8ビット操作）あるいはAX（16ビット操作）のレジスタから，8あるいは16ビットのイミディエイト・データの減算を行なう．結果はフラグを設定するために用いられてから捨てられる．

2つのストリング・プリミティブ命令，ＣＭＰＳとＳＣＡＳも比較動作を行なう．これらの命令については，この章の後で他のプリミティブ命令と共に述べる．

(1) ストリングの長さの計算

図4-24のルーチンは，ストリング中のキャラクタの数を決める．

このルーチンは，ＳＩレジスタが調べられるストリングのアドレスを示し，ＡＨがストリングの終わりを示すキャラクタを含むと仮定している．このルーチンが実行を終えたとき，ＤＸレジスタはストリングの先頭と終了キャラクタの間のキャラクタ数を含んでいる．

```
                        MOV     DX,0FFFFH               ;INITIALIZE COUNT TO -1
SCAN$FOR$DELIMITER:     INC     DX                      ;INCREMENT COUNT
                        MOV     AL,[SI]                 ;LOAD BYTE FROM STRING
                        INC     SI                      ;UPDATE POINTER
                        CMP     AH,AL                   ;COMPARE WITH TERMINATION
                        JNZ     SCAN$FOR$DELIMITER      ;BRANCH IF NOT TERMINATION
                        RET
```

図4-24　ストリングの長さの計算

ストリング・プリミティブ命令ＳＣＡＳは，このルーチンのメモリの減少と実行を速めるために用いることができる．ＳＣＡＳ命令は，この章の後で他のストリング・プリミティブ命令と共に述べる．

(2) 系列中で最大の8ビット符号なし数値を求める

図4-25のルーチンは，8ビット符号なし数値の系列中で最大値を決定する．このルーチンは，ＳＩレジスタが調べられる一連の数値のアドレスを示し，ＣＸレジスタが調べられるバイト数を含むと仮定している．このルーチンの実行が終了したとき，ＡＨは最大値を含み，ＤＸは最大値の位置を示している．

```
                        XOR     AH,AH                   ;INITIALIZE MAX. NUMBER
SCAN$NEXT$BYTE:         MOV     AL,[SI]                 ;LOAD BYTE FROM SEQUENCE
                        CMP     AH,AL                   ;COMPARE WITH CURRENT MAX. #
                        JAE     UPDATE$PTR
                        MOV     AH,AL                   ;SAVE NEW MAX. NUMBER
                        MOV     DX,SI                   ;SAVE LOCATION OF MAX. #
UPDATE$PTR:             INC     SI
                        DEC     CX
                        JNZ     SCAN$NEXT$BYTE
                        RET
```

図4-25　最大の8ビット数値を求める

図4-25と図4-26のルーチンは，ストリング・プリミティブ命令を用いることによって改善できる．

(3) 系列中で最大の16ビット数値を求める

図4-26のルーチンは，16ビット符号付き数値の系列中で最大値を決定する．このルーチンは，ＳＩレジスタが調べられる数値列のアドレスを示し，ＣＸレジスタが調べられるワード数を含むと仮定している．

```
                MOV     BX,8000H            ;INITIALIZE MAX. NUMBER
SCAN$LOOP:      MOV     AX,[SI]             ;LOAD NUMBER FROM SEQUENCE
                CMP     BX,AX               ;COMPARE WITH CURRENT MAX. NUMBER
                JGE     UPDATE$PTR
                MOV     BX,AX               ;SAVE NEW MAX. NUMBER
                MOV     DX,SI               ;SAVE LOCATION OF MAX. NUMBER
UPDATE$PTR:     INC     SI                  ;UPDATE PRT.
                INC     SI
                DEC     CX                  ;DECREMENT AND TEST FOR DONE
                JNZ     SCAN$LOOP
                RET
```

図4-26　最大の16ビット数値を求める

(4) バッファの変換

この章の初めに，２つのバッファ変換ルーチンを示した．次のルーチンはエラーのチェックを含んでいる．変換されるバッファ内のキャラクタは，20_{16}から５F_{16}までの範囲になければならない．このルーチンは，ＢＸレジスタが変換テーブルのアドレスを含み，ＳＩレジスタが変換されるバッファのアドレスを含み，ＣＸレジスタが変換されるデータ要素の数を含むと仮定している．

```
TRANSLATE$LOOP:   MOV    AL,[SI]             ;LOAD BYTE FROM SOURCE
                  SUB    AL,20H              ;NORMALIZE
                  JB     TRANSLATE$ERROR     ;IF LESS THAN 0, REPORT ERROR
                  CMP    AL,3FH              ;COMPARE WITH NORMALIZED MAX.
                  JA     TRANSLATE$ERROR     ;IF GREATER, REPORT ERROR
                  XLAT                       ;TRANSLATE NORMALIZED VALUE
                  MOV    [SI],AL             ;STORE CONVERTED DATA
                  INC    SI                  ;ADJUST POINTERS
                  DEC    CX
                  JNZ    TRANSLATE$LOOP
                  RET                        ;GOOD RETURN WITH Z=1
TRANSLATE$ERROR:  RET                        ;ERROR RETURN WITH Z=0
```

図4-27　範囲をチェックするバッファ内容の変換

変換エラーがなければＺＦ＝１で，１つあるいはそれ以上の変換エラーがあればＺＦ＝０で戻る．減算命令によって変換テーブルの大きさが40_{16}バイトに制限されていることに注意．２つのＣＭＰ命令を変換テーブルの20_{16}バイト前の位置を示すＢＸレジスタと用いれば，データを有効に利用することができる．

4.3 論理演算命令

8086は，次に示す通常の論理演算機能を持つ．

ＡＮＤ

ＮＯＴ

ＯＲ

ＸＯＲ

さらに，オペランドのどちらも変えないAND操作を行なうTEST命令がある．

8086の論理演算命令を**表4-7**に示す．

図4-29は，8086の論理演算の例を示している．次の情況を考える．

1．I/Oポートは，一連のデータ・ブロックを受け取る．このデータ・ブロックは，Signetics 16進コードである．

2．I/Oポートは，キャラクタが有効になるたびに割り込みを発生する．

図4-29のルーチンは，次のようなステップで上記の情況を処理するインタラプト・サービス・ルーチンである．

1．情報をそのままセーブする．

2．その情報をオブジェクト・コードに変換する．

3．チェックサムを調べる．

4．処理が終了すれば，メッセージの“メッセージ完了”ビットをセットする．

図4-29のルーチンは，図4-28のフローチャートを実行して，上記機能を行なっている．

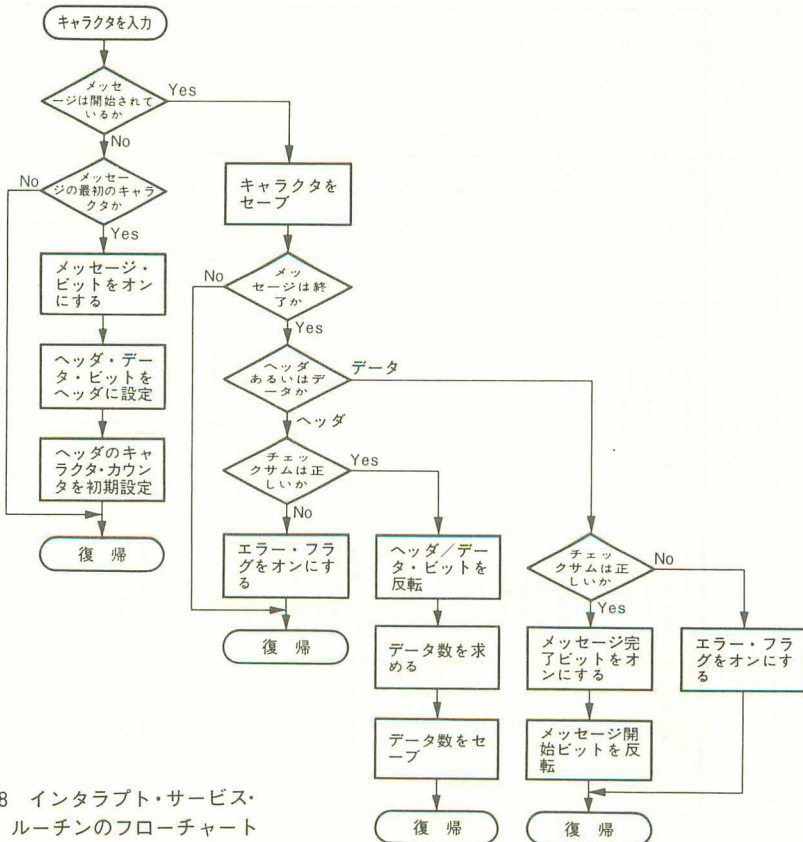



図4-28　インタラプト・サービス・ルーチンのフローチャート

表 4-7　8086の論理演算命令

ニーモニック	オペランド	オブジェクト・コード	バイト	クロック	ステータス O	D	I	T	S	Z	A	P	C	動作内容
AND	mem/reg₁, mem/reg₂	001000dw mod rrr r/m (DISP) (DISP)	2, 3または4	reg-reg: 3 mem-reg: 9 + EA reg-mem: 16 + EA	X				X	X	?	X	X	〔mem/reg₁〕←〔mem/reg₁〕AND〔mem/reg₂〕 mem/reg₁で示されるメモリ位置あるいはレジスタの8あるいは16ビットの内容と，mem/reg₂で示されるメモリ位置あるいはレジスタの8あるいは16ビットの内容とのアンドをとり，mem/reg₁で示されるメモリ位置あるいはレジスタに結果を残す．
AND	mem/reg, data	1000000w mod 100 r/m (DISP) (DISP) kk jj (w=1のとき)	3, 4, 5または6	reg: 4 mem: 17 + EA	X				X	X	?	X	X	〔mem/reg〕←〔mem/reg〕AND data mem/regで示されるメモリ位置あるいはレジスタの8あるいは16ビットの内容と，8あるいは16ビットのイミディエイト・データとのアンドをとり，mem/regで示されるメモリ位置あるいはレジスタに結果をストアする．
AND	ac,data	0010010w kk jj (w=1のとき)	2または3	4	X				X	X	?	X	X	〔ac〕←〔ac〕AND data AL（8ビット操作）あるいはAX（16ビット操作）のレジスタと，8あるいは16ビットのイミディエイト・データとのアンドをとり，結果をALあるいはAXのレジスタに残す．
NOT	mem/reg	1111011w mod 010 r/m (DISP) (DISP)	2, 3または4	reg: 3 mem: 16 + EA										〔mem/reg〕←$\overline{〔mem/reg〕}$ mem/regで示されるメモリ位置あるいはレジスタの8あるいは16ビットの内容の1の補数をとる．
OR	mem/reg₁, mem/reg₂	000010dw mod rrr r/m (DISP) (DISP)	2, 3または4	reg-reg: 3 mem-reg: 9 + EA reg-mem: 16 + EA	X				X	X	?	X	X	〔mem/reg₁〕←〔mem/reg₁〕OR〔mem/reg₂〕 mem/reg₁で示されるメモリ位置あるいはレジスタの8あるいは16ビットの内容と，mem/reg₂で示されるメモリ位置あるいはレジスタの8あるいは16ビットの内容とのオアをとり，mem/reg₁で示されるメモリ位置あるいはレジスタに結果を残す．
OR	mem/reg, data	1000000w mod 001 r/m (DISP) (DISP) kk jj (w=1のとき)	3, 4, 5または6	reg: 4 mem: 17 + EA	X				X	X	?	X	X	〔mem/reg〕←〔mem/reg〕OR data mem/regで示されるメモリ位置あるいはレジスタの8あるいは16ビットの内容と，8あるいは16ビットのイミディエイト・データとのオアをとり，mem/regで示されるメモリ位置あるいはレジスタに結果を残す．

表4-7　8086の論理演算命令(続き)

ニーモニック	オペランド	オブジェクト・コード	バイト	クロック	ステータス O	D	I	T	S	Z	A	P	C	動作内容
OR	ac,data	0000110w kk jj (w=1のとき)	2または3	4	X				X	X	?	X	X	〔ac〕←〔ac〕OR data AL (8ビット操作) あるいはAX (16ビット操作) のレジスタと，8あるいは16ビットのイミディエイト・データとのオアをとり，ALあるいはAXのレジスタに結果を残す．
TEST	mem/reg$_1$, mem/reg$_2$	1000010w mod rrr r/m (DISP) (DISP)	2, 3または4	reg-reg: 3 mem-reg: 9 + EA	X				X	X	?	X	X	〔mem/reg$_1$〕AND〔mem/reg$_2$〕 mem/reg$_1$で示されるメモリ位置あるいはレジスタの8あるいは16ビットの内容と，mem/reg$_2$で示されるメモリ位置あるいはレジスタの8あるいは16ビットの内容とのアンドをとる．結果はフラグを設定するために用いられてから捨てられる．
TEST	mem/reg, data	1111011w mod 000 r/m (DISP) (DISP) kk jj (w=1のとき)	3, 4, 5または6	reg: 5 mem: 11 + EA	X				X	X	?	X	X	〔mem/reg〕AND data mem/regで示されるメモリ位置あるいはレジスタの8あるいは16ビットの内容と，8あるいは16ビットのイミディエイト・データとのアンドをとる．結果はフラグを設定するために用いられてから捨てられる．
TEST	ac,data	1010100w kk jj (w=1のとき)	2または3	4	X				X	X	?	X	X	〔ac〕AND data AL (8ビット操作) あるいはAX (16ビット操作) のレジスタと，8あるいは16ビットのイミディエイト・データとのアンドをとる．結果はフラグを設定するために用いられてから捨てられる．
XOR	mem/reg$_1$, mem/reg$_2$	001100dw mod rrr r/m (DISP) (DISP)	2, 3または4	reg-reg: 3 mem-reg: 9 + EA reg-mem: 16 + EA	X				X	X	?	X	X	〔mem/reg$_1$〕←〔mem/reg$_1$〕XOR〔mem/reg$_2$〕 mem/reg$_1$で示されるメモリ位置あるいはレジスタの8あるいは16ビットの内容と，mem/reg$_2$で示されるメモリ位置あるいはレジスタの8あるいは16ビットの内容とのエクスクルーシブ・オアをとり，結果はmem/reg$_1$で示されるメモリ位置あるいはレジスタに残す．
XOR	mem/reg, data	1000000w mod 110 r/m (DISP) (DISP) kk jj (w=1のとき)	3, 4, 5または6	reg: 4 mem: 17 + EA	X				X	X	?	X	X	〔mem/reg〕←〔mem/reg〕XOR data mem/regで示されるメモリ位置あるいはレジスタの8あるいは16ビットの内容と，8あるいは16ビットのイミディエイト・データとのエクスクルーシブ・オアをとり，結果はmem/regで示されるメモリ位置あるいはレジスタに残す．
XOR	ac,data	0011010w kk jj (w=1のとき)	2または3	4	X				X	X	?	X	X	〔ac〕←〔ac〕XOR data AL (8ビット操作) あるいはAX (16ビット操作) のレジスタと8あるいは16ビットのイミディエイト・データとのエクスクルーシブ・オアをとり，結果をALあるいはAXのレジスタに残す．

以下に示すように，Signetics オブジェクト・コードは次の要素を持つ．

1．スペースを含む，任意数のプリントされないキャラクタによるギャップ．
2．ブロック・キャラクタの開始，すなわち1個のコロン．
3．アドレス・フィールド，すなわち4個の16進キャラクタ．
4．カウント・フィールド，すなわち0から$1E_{16}$の範囲の2個の16進キャラクタ．
5．アドレスとカウントのフィールドのブロック・コントロール・キャラクタ，すなわち2つの16進キャラクタ．
6．データ・フィールド．このブロックによってロードされるメモリの数を表わすカウント・フィールドの値の2倍の長さを持つ．
7．ブロック・コントロール・キャラクタ．これは，2つの16進キャラクタである．

Signeticsオブジェクト・コード・フォーマットの例

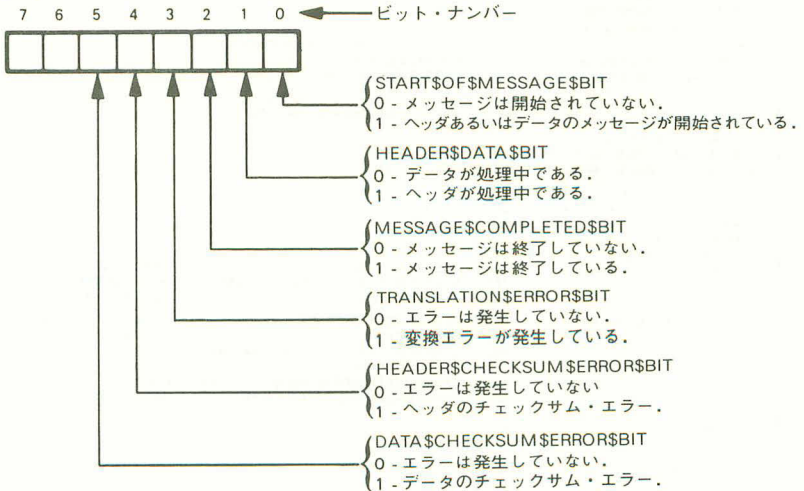
② ― ブロック・キャラクタの開始(コロン)
③ ― ブロックの開始アドレス(H'0500')
④ ― ブロックのバイト数(H'0A'＝10)
⑤ ― フィールド3と4のBCCバイト(H'3C')
⑥ ― データ、1バイトにつき2バイト
⑦ ― フィールド6のBCCバイト(H'30')

ブロック・コントロール・キャラクタ，あるいはチェックサムとして知られているものは，データ・ストリング中のデータ・バイトを操作して作られる．Signetics オブジェクト・コードのフォーマットは，2つのブロック・コントロール・キャラクタを含む．第1のブロック・コントロール・キャラクタ⑤は，ブロックの先頭アドレスとブロック中のバイト数を含む，前の3つのデータ・バイトに適用される．第2のブロック・コントロール・キャラクタ⑦は，フィールド⑥のデータ・バイトのストリングに適用される．

ブロック・コントロール・キャラクタを生成するために，まずキャラクタがクリヤされ，次に関連のあるストリング中の各データ・バイトとのエクスクルーシブＯＲが繰り返し行なわれる．エクスクルーシブＯＲをとるたびに，結果を1ビット左へローテートする．ブロック・コントロール・キャラクタの理論を示すために，Signetics オブジェクト・コードのフォーマットの第1のブロック・コントロール・キャラクタを考える．一連のバイト 05　00　0A　から次のように生成が行なわれる．

BCC	Data	BCC XOR Data	左へローテート
00000000	00000101	00000101	00001010
00001010	00000000	00001010	00010100
00010100	00001010	00011110	00111100
			BCC = 3CH

図4-29のルーチンでは，データがストアされるバッファ以外に，メモリに位置する次の4つの変数を用いている．

STATUS$BYTE:

CHARACTER$COUNT: ヘッダあるいはデータのブロックに読まれずに残っているキャラクタ数を表わす1バイト．
メッセージの開始によってこの変数は，NUMBEROFHEADER$CHARACTER，すなわち8に初期設定される．ヘッダが処理されると，この変数は2*(データ・ブロックのオブジェクト・コードのバイト数)＋2に初期設定される．

OBJECT$BYTE$COUNT: データ・ブロック中のオブジェクト・コードのバイト数を表わす1バイト．
この変数は，ヘッダの処理が終わった後に初期設定される．

BUFFER$POINTER: I/Oポートからのデータの次のバイトがストアされるべき位置を示す16ビットのオフセット・アドレス．この変数は，メッセージの開始によって初期設定され，イミディエイト・データがロードされる．これは，バッファが常に固定された位置にあることを仮定している．もしバッファの位置を変える必要があるならば，この変数はシステムあるいはユーザ・プログラムによって初期設定できる．

```
;
;  This page contains the equates for this program. As mentioned
;  earlier, equates allow descriptive names to be used in a program.
;
;
;    STATUS BYTE EQUATES
START$OF$MESSAGE$BIT           EQU   01H
HEADER$DATA$BIT                EQU   02H
MESSAGE$COMPLETED$BIT          EQU   04H
TRANSLATION$ERROR$BIT          EQU   08H
HEADER$CHECKSUM$ERROR$BIT      EQU   10H
DATA$CHECKSUM$ERROR$BIT        EQU   20H
;    BUFFER ADDRESS EQUATES
BUFFER$ADDRESS                 EQU   1000H;OFFSET ADDRESS FOR BUFFER
START$OF$HEADER$POINTER        EQU   BUFFER$ADDRESS
START$OF$DATA$POINTER          EQU   BUFFER$ADDRESS + 8
;    I/O EQUATE
CHARACTER$PORT                 EQU   10H;I/O PORT ADDRESS FOR DATA
;    MISCELLANEOUS EQUATES
START$OF$MESSAGE$CHARACTER     EQU   3AH
NUMBER$OF$HEADER$CHARACTERS    EQU   08H
;    DATA DEFINITION
STATUS$BYTE                    DB    1
CHARACTER$COUNT                DB    1
BUFFER$POINTER                 DW    1
OBJECT$BYTE$COUNT              DB    1
INTERRUPT$HANDLER              IN    AL,CHARACTER$PORT                    ;READ CHARACTER
                               TEST  STATUS$BYTE,START$OF$MESSAGE$BIT     ;HAS A MESSAGE
                               JNZ   HEADER$OR$DATA                       ;BEEN STARTED?

START$OF$MESSAGE$CODE          CMP   AL,START$OF$MESSAGE$CHARACTER        ;START OF MESSAGE
                               JNZ   PERFORM$A$RET                        ;CHARACTER?
                               OR    STATUS$BYTE,START$OF$MESSAGE$BIT  OR
                                     HEADER$DATA$BIT                      ;INITIALIZE
                               MOV   CHARACTER$COUNT,NUMBER$OF$HEADER$CHARACTERS
                               MOV   BUFFER$POINTER,BUFFER$ADDRESS        ;MOVE IMMEDIATE

PERFORM$A$RET                  RET                                        ;DATA TO BUFFER
                                                                          ;POINTER
HEADER$OR$DATA                 MOV   DI,BUFFER$POINTER                    ;SAVE CHARACTER
                               MOV   [DI],AL
                               INC   DI                                   ;UPDATE POINTER
                               MOV   BUFFER$POINTER,DI
                               DEC   CHARACTER$COUNT                      ;DECREMENT AND TEST
                               JNZ   PERFORM$A$RET                        ;FOR MESSAGE DONE
                               TEST  STATUS$BYTE,HEADER$DATA$BIT
                               JZ    DATA$PROCESSING
HEADER$PROCESSING              MOV   CX,0004                              ;SET UP FOR ASCII TO HEX
                               MOV   SI,START$OF$HEADER$POINTER           ;CONVERSION
                               MOV   DI,SI
HEADER$TRANSLATE$LOOP          CALL  CONVERT$TWO$ASCII$TO$HEX             ;CONVERT TWO ASCII CHARACTERS
                               JZ    TRANSLATION$ERROR                    ;TO ONE HEX BYTE
                               MOV   [DI],AL
                               INC   DI
                               DEC   CX                                   ;DECREMENT AND TEST FOR DONE
                               JNZ   HEADER$TRANSLATE$LOOP

                               MOV   SI,START$OF$HEADER$POINTER           ;SET UP FOR HEADER CHECKING
                               XOR   AX,AX
                               MOV   CX,0003
HEADER$CHECKSUM$LOOP           XOR   AL,[SI]                              ;CALCULATE BLOCK CHECKSUM
                               ROL   AL,1                                 ;FROM CHARACTERS
                               INC   SI

                               DEC   CX                                   ;DECREMENT AND TEST
                               JNZ   HEADER$CHECKSUM$LOOP                 ;FOR CHECKSUM DONE
                               CMP   AL,[SI]                              ;COMPARE CALCULATED CHECKSUM
                               JNZ   HEADER$CHECKSUM$ERROR                ;WITH RECEIVED CHECKSUM
                               XOR   STATUS$BYTE,HEADER$DATA$BIT          ;HEADER GOOD, SWITCH TO
                                                                          ;  DATA PROCESSING
                               MOV   AX,[SI-2]                            ;LOAD # OF OBJECT
                               MOV   OBJECT$BYTECOUNT,AX                  ;CODE BYTES FROM HEADER
                               SAL   AX,1                                 ;GET NUMBER OF ASCII CHARACTERS
                               ADD   AX,02                                ;ADD 2
                               RET   CHARACTER$COUNT,AX                   ;SAVE FOR DATA PROCESSING
HEADER$CHECKSUM$ERROR          MOV   STATUS$BYTE,HEADER$CHECKSUM$ERROR$BIT
                                                                          ;TURN ON ERROR BIT
                               RET
```

図4-29　インタラプト・サービス・ルーチン

```
TRANSLATION$ERROR       MOV   STATUS$BYTE,TRANSLATION$ERROR$BIT      ;TURN ON ERROR BIT
                        RET
DATA$PROCESSING         MOV   CX,OBJECT$BYTE$COUNT                   ;SET UP FOR CONVERSION
                        MOV   SI,START$OF$DATA$POINTER               ;FROM ASCII TO HEX
                        MOV   DI,START$OF$DATA$POINTER-4
DATA$TRANSLATE$LOOP     CALL  CONVERT$TWO$ASCII$TO$HEX
                        JZ    TRANSLATION$ERROR
                        MOV   [DI],AL
                        INC   DI
                        DEC   CX                                     ;DECREMENT AND TEST FOR
                        JNZ   DATA$TRANSLATE$LOOP                    ;DONE
                        MOV   SI,START$OF$DATA$POINTER -4            ;SET UP FOR CHECKSUM
                        XOR   AX,AX                                  ;CALCULATION
                        MOV   CX,OBJECT$BYTE$COUNT
DATA$CHECKSUM$LOOP      XOR   AL,[SI]
                        ROL   AL,1                                   ;CALCULATE CHECKSUM
                        INC   SI
                        DEC   CX
                        JNZ   DATA$CHECKSUM$LOOP
                        CMP   AL,[SI]                                ;COMPARE CALCULATED CHECKSUM
                        JNZ   DATA$CHECKSUM$ERROR                    ;WITH RECEIVED CHECKSUM
                        XOR   STATUS$BYTE,START$OF$MESSAGE$BIT       ;TURN ON
                              OR MESSAGES$COMPLETED$BIT              ;MESSAGE COMPLETED BIT
                        RET                                          ;TURN OFF START OF MESSAGE BIT
DATA$CHECKSUM$ERROR     MOV   STATUS$BYTE,DATA$CHECKSUM$ERROR$BIT    ;TURN ON ERROR BYTE
                        RET
```

図4-29　インタラプト・サービス・ルーチン(続き)

図4-28に示すプログラムの理論では，いくつかの仮定がある．この仮定には次のものが含まれる．

1．ＣＰＵの状態は退避されていて，復帰の際に正しく復元される．
2．セグメント・レジスタは正しい値に設定されている．
3．I/Oポートによって設定されるハードウエアのエラー・フラグは，他で処理される．
4．CONVERTTWOASCIITOHEXという名前のサブルーチンは，ＤＩレジスタで示される２つのＡＳＣＩＩキャラクタを16進バイトに変換して，その値をＡＬに設定して戻る．

ある種類のシステム・インタラプト処理プログラムが，最初の３つの条件を備えていることを期待するのは当然といえる．もしそうでなければ，上記条件は次のようにして処理できる．

1．ＣＰＵ状態の退避・復元には，図4-14と図4-15に示した命令を用いる．ただし，このルーチンはＢＸ，ＤＸ，あるいはＢＰのレジスタを用いていないことに注意．したがって，これらのレジスタは退避・復元の必要がない．ＲＥＴ命令はＩＲＥＴ命令に変換しなければならない．
2．適当なセグメント・レジスタ初期設定を行なう．
3．入力装置のステータス・ポートを読み出して，エラー表示のためにステータス・バイトのフラグを設定する．ビット６と７はこの目的のために用いられる．

CONVERTTWOASCIITOHEX のルーチンは，この章の後で述べる．

4.4 ストリング・プリミティブ命令

8086のストリング・プリミティブ命令を**表4-8**に示す．各々のストリング・プリミティブ命令は，通常，命令ループで処理される一連の操作を行なう．ストリング・プリミティブ命令は，指定された1つの操作を行ない，次にこの操作に含まれているポインタ・レジスタの増減を行なう．各繰り返しごとに，影響を受けるポインタ・レジスタは，1あるいは2の増減が行なわれる．ポインタ・レジスタは，ディレクション・フラグが0ならば増加し，1ならば減少する．ストリング・プリミティブ・オペレーション・コードの最下位ビットが0ならば，ポインタ・レジスタは1の増減が行なわれる．ストリング・プリミティブ・オペレーション・コードの最下位ビットが1ならば，影響を受けるポインタ・レジスタは2の増減が行なわれる．

次の5つのストリング・プリミティブ命令が存在する．

MOVS－メモリからメモリへの8あるいは16ビットのデータを移動する．

LODS－メモリから8あるいは16ビットのデータをALあるいはAXのレジスタにロードする．

STOS－AL（8ビット操作）あるいはAX（16ビット操作）のレジスタの内容をメモリにストアする．

SCAS－メモリとAL（8ビット操作）あるいはAX（16ビット操作）のレジスタの比較を行なう．

CMPS－メモリとメモリの比較を行なう．

ストリング・プリミティブ命令は，次に示すような固定されたアドレッシング・モードを用いる．

MOVS－データ・セグメントのSIレジスタで示されるメモリ位置から，エキストラ・セグメントのDIレジスタで示されるメモリ位置にデータを移動する．

LODS－データ・セグメントのSIレジスタで示されるメモリ位置から，ALあるいはAXのレジスタにデータをロードする．

STOS－ALあるいはAXのレジスタの内容を，エキストラ・セグメントのDIレジスタで示されるメモリ位置にストアする．

SCAS－エキストラ・セグメントのDIレジスタで示されるメモリ位置のデータと，ALあるいはAXのレジスタの内容を比較する．

CMPS－データ・セグメントのSIレジスタで示されるメモリ位置のデータと，エキストラ・セグメントのDIレジスタで示されるメモリ位置のデータを比較する．

セグメント変更プレフィックスによって，SIでデータ・セグメント以外のセグメントにアクセスすることができる．セグメント変更プレフィックスは，DIレジスタには用いられない．DIレジスタは，エキストラ・セグメントをアクセスしなければならない．

表 4-8 ストリング・プリミティブ命令

ニーモニック	オペランド	オブジェクト・コード	バイト	クロック	ステータス O	D	I	T	S	Z	A	P	C	動作内容
LODS		1010110w	1	12 9 + 13*										〔ac〕←〔〔SI〕〕, 〔SI〕←〔SI〕± DELTA SIで示されるメモリ位置から, AL (8ビット操作) あるいはAX (16ビット操作) のレジスタにデータを移動する. ディレクション・フラグの値に依存して, SIの増減を行なう. DELTA は, w=0のときは1で, w=1のときは2である.
MOVS		1010010w	1	18 9 + 17*										〔〔DI〕〕←〔〔SI〕〕, 〔SI〕←〔SI〕± DELTA 〔DI〕←〔DI〕± DELTA SIで示されるメモリ位置からDIで示されるメモリ位置へ, 8あるいは16ビットのデータを移動する. ディレクション・フラグの値に依存して, SIとDIの増減を行なう. DELTA は, w=0のときは1で, w=1のときは2である.
STOS		1010101w	1	11 9 + 10*										〔〔DI〕〕←〔ac〕, 〔DI〕←〔DI〕± DELTA AL (8ビット操作) あるいはAX (16ビット操作) のレジスタの内容を, DIレジスタで示されるメモリ位置に移動する. ディレクション・フラグの値に依存して, DIの増減を行なう. DELTA は, w=0のときは1で, w=1のときは2である.
CMPS		1010011w	1	22 9 + 22*	X				X	X	X	X	X	〔〔SI〕〕−〔〔DI〕〕, 〔SI〕←〔SI〕± DELTA 〔DI〕←〔DI〕± DELTA SIレジスタで示されるアドレスの8あるいは16ビットの内容から, DIレジスタで示されるアドレスの8あるいは16ビットの内容を減じる. 結果はフラグを設定するために用いてから捨てられる. ディレクション・フラグの値に依存して, SIとDIの増減を行なう. DELTAは, w=0のときは1で, w=1のときは2である.
SCAS		1010111w	1	15 9 + 15*	X				X	X	X	X	X	〔ac〕−〔〔DI〕〕 AL (8ビット操作) あるいはAX (16ビット操作) のレジスタから, DIレジスタで示されるアドレスの8あるいは16ビットの内容を減じる. 結果はフラグを設定するために用いてから捨てられる. ディレクション・フラグの値に依存して, DIの増減を行なう. DELTAは, w=0のときは1で, w=1のときは2である.

*REPプレフィックスが先行した場合, 最初の転送は*の数に9を加えたクロックに, 以後の転送は*の数のクロックになる.

4.4.1 REPプレフィックス

ＲＥＰは，ストリング・プリミティブ命令を再実行ループに変換する１バイトのプレフィックスである．

ストリング・プリミティブ命令は，それぞれループの１回の繰り返しとして実行される．ソースとディスティネーションのポインタ・レジスタのＳＩとＤＩは，ストリング・プリミティブ命令で，ソースやディスティネーションのメモリ・アドレスを与えると仮定されている．このアドレスは，ストリング・プリミティブ命令の実行に続いて，自動的に増加あるいは減少する．これにより，アドレスは，アクセスされるストリングの次の位置を示す．ＲＥＰプレフィックスは，終結条件が成立するまでストリング・プリミティブ命令の実行を続けさせる条件を指定する．

ＭＯＶＳ，ＬＯＤＳ，ＳＴＯＳのストリング・プリミティブについては，１つの終結条件が存在する．ＣＸレジスタはカウンタとして取り扱われ，ストリング・プリミティブ命令が実行されるたびに，ＣＸレジスタの内容はＲＥＰプレフィックスによって自動的に減少する．ＣＸレジスタの内容が０に減少すると，ストリング・プリミティブに続く命令が実行される．

ＣＭＰＳとＳＣＡＳもまた，ＲＥＰプレフィックスが存在するとき，ＣＸレジスタをカウンタとして用い，ＭＯＶＳ，ＬＯＤＳ，ＳＴＯＳについて述べたように，ＣＸレジスタが減少して０になることが終結条件となる．さらに，ＣＭＰＳとＳＣＡＳは，実行を繰り返すたびにステータス・フラグを設定する．ゼロ・フラグの値は，別の終結条件として用いられる．これを可能とするために，ＣＭＰＳとＳＣＡＳのストリング・プリミティブは，ＲＥＰプレフィックスの次の２つの形式を用いる．

1．ＲＥＰＺあるいはＲＥＰＥ．これは，ストリング・プリミティブの繰り返される実行で，ゼロ・フラグがリセットされれば終了する．
2．ＲＥＰＮＺあるいはＲＥＰＮＥ．これは，繰り返される実行で，ゼロ・フラグが，セットされれば終了する．

要するに，ＲＥＰプレフィックスは，ストリング・プリミティブ命令の実行に，次のステップを加える．

1．ＣＸレジスタの内容を調べる．ＣＸが０ならば，ストリング・プリミティブに続く命令に進む．
2．保留となっている割り込みを処理する．
3．ストリング・プリミティブ命令を一度実行する．ポインタ・レジスタのアドレスは，ストリング・プリミティブ命令実行の通常の部分として，このステップの間に増加あるいは減少する．
4．ＣＸレジスタの内容を減じる．

5a．ＭＯＶＳ，ＬＯＤＳ，あるいはＳＴＯＳについてはステップ１に進む．

5b．ＣＭＰＳあるいはＳＣＡＳについては，ゼロ・フラグとＲＥＰプレフィックスで指定される条件を比較する．指定されたゼロ・フラグの状態でなければ，ステップ１に

戻り，そうでなければストリング・プリミティブに続く命令を実行する．

ストリング・プリミティブ命令は非常に有力である．

```
MOV  AL,〔SI〕
INC  SI
```

あるいは，

```
MOV  AX,〔SI〕
INC  SI
INC  SI
```

などの命令は，

```
MOVSB
```

あるいは，

```
MOVSW
```

で置き換えられる．

図4-2のルーチンを考える．もしディレクション・フラグが0に設定されていれば，その一連の命令は，

```
REP  MOVSW
     RET
```

で置き換えられる．さらに，REPとMOVSWの命令が1バイト命令であることに注意すれば，このルーチンをコールすることの方が，プログラム中に直接

```
REP  MOVSW
```

を挿入するよりも損失が大きくなる．

図4-9のバッファ初期設定ルーチンは，

```
REP  STOSB
```

で置き換えられる．この置き換えでは，ディレクション・フラグが0に設定されていることを仮定している．

練習に，読者は，この章の初めに示した他のプログラムを調べて，ストリング・プリミティブの使用が可能な例を見つけられたい．

バイトより成る2つのストリングを比較する，図4-1に示されているプログラムの変形を考える．完全に書けば，プログラムは次のようになる．

```
COMP$BYTES:   MOV   AL,[SI]        ;LOAD BYTE FROM SOURCE
              CMP   [DI],AL        ;COMPARE WITH DESTINATION
              JZ    EQUAL          ;TEST FOR SIMILAR BYTES
              INC   SI             ;ADJUST POINTERS
              INC   DI
              DEC   CX             :DECREMENT NUMBER TO MOVE
              JNZ   COMP$BYTES     ;LOOP IF NOT DONE
```

図4-30　8ビットのバッファとバッファの比較

ＣＭＰは，レジスタとレジスタ，レジスタとメモリ，あるいはメモリとレジスタで，バイトあるいはワードの比較を行なう．**図4-30**では，同一のバイトを探して，２つのメモリ・バッファの比較が行なわれる．ポインタ・レジスタのＳＩとＤＩは，それぞれソースとディスティネーションのアドレスを示す．レジスタのＳＩ，ＤＩ，さらにＣＸは，ＣＭＰＳＢとＣＭＰＳＷのストリング・プリミティブで用いられるレジスタなので，図4-30では慎重に選択されている．この結果，図4-30に示すプログラムの全体を次のもので置き換えることが可能となる．

```
REPNZ    CMPSB
JZ       EQUAL
```

図4-31　別の８ビットのバッファとバッファの比較

4.5　プログラム・カウンタ制御命令

8086の命令のこのグループは，無条件にプログラム・カウンタの内容を変え，さらにある場合には，コード・セグメントの内容も同様に変更する．命令の要約を**表４-9**に示す．

ＣＡＬＬ命令は，プログラムからサブプログラムに制御を移すために用いられる．サブプログラムは，カレント・コード・セグメントあるいは命令で指定されるコード・セグメントに存在する．サブプログラムのアドレスは，命令でイミディエイト・アドレスとして与えられるか，あるいはメモリまたはレジスタにストアされている．8086には，次の４つの可能なＣＡＬＬ命令がある．

	カレント・コード・セグメント	命令指定の コード・セグメント
イミディエイト・アドレス	CALL disp16	CALL addr
メモリあるいはレジスタ内のアドレス	CALL mem/reg	CALL mem

ＣＡＬＬ　disp16　は，符号付き16ビット数値がプログラム・カウンタに加算されるただ１つの命令である．他の３つのＣＡＬＬ命令は，メモリあるいはレジスタからプログラム・カウンタに直接データを移動する．以前のコード・セグメントやプログラム・カウンタの内容はスタックにプッシュされて，退避される．

RETURN 命令は，サブプログラムからそれを呼び出したプログラムに制御を戻すために用いられる．サブプログラムを終結させるRETURN 命令は，サブプログラムを呼び出すＣＡＬＬの逆の操作を行なう．RETURN 命令が実行されると，プログラム・カウンタと必要ならばコード・セグメント・レジスタに，スタックからデータがポップされる．さらに，RETURN 命令は，スタック・ポインタにディスプレイスメントを加えることもでき

表4-9　プログラム・カウンタ制御命令

ニーモニック	オペランド	オブジェクト・コード	バイト	クロック	ステータス									動　作　内　容
					O	D	I	T	S	Z	A	P	C	
CALL	addr	9A kk jj hh gg	5	28										〔SP〕←〔SP〕－2，〔〔SP〕〕←〔PC〕， 〔SP〕←〔SP〕－2，〔〔SP〕〕←〔CS〕， 〔PC〕←addr（オフセット部），〔CS〕←addr（セグメント部） 他のコード・セグメント内のサブルーチンをコールする．新しいオフセット・アドレスはjjkkに，新しいセグメント・アドレスはgghhになる．
CALL	disp16	E8 kk jj	3	19										〔SP〕←〔SP〕－2，〔〔SP〕〕←〔PC〕 〔PC〕←〔PC〕＋disp16 現在のコード・セグメント内のサブルーチンをコールする．
CALL	mem (SEG ＋PC)	FF mod 011 r/m (DISP) (DISP)	2，3または4	37＋EA										〔SP〕←〔SP〕－2，〔〔SP〕〕←〔PC〕， 〔SP〕←〔SP〕－2，〔〔SP〕〕←〔CS〕， 〔PC〕←〔mem〕，〔CS〕←〔mem＋2〕 他のコード・セグメント内のサブルーチンをコールする．memで示されるメモリ位置の16ビットの内容がPCに移動し，続くメモリ・ワードの内容がCSレジスタにロードされる．
CALL	mem/reg （PCのみ）	FF mod 010 r/m (DISP) (DISP)	2，3または4	21＋EA (mem) 16 (reg)										〔SP〕←〔SP〕－2，〔〔SP〕〕←〔PC〕 〔PC〕←〔mem/reg〕 現在のコード・セグメント内のサブルーチンをコールする．mem/regで示されるメモリ位置あるいはレジスタの16ビットの内容がPCに移動する．
RET		C3	1	16										〔PC〕←〔〔SP〕〕，〔SP〕←〔SP〕＋2 現在のコード・セグメント内の呼び出し元プログラムにリターンする．
RET		CB	1	24										〔PC〕←〔〔SP〕〕，〔SP〕←〔SP〕＋2， 〔CS〕←〔〔SP〕〕，〔SP〕←〔SP〕＋2 他のコード・セグメントの呼び出し元プログラムにリターンする．
RET	disp16	C2 kk jj	3	20										〔PC〕←〔〔SP〕〕，〔SP〕←〔SP〕＋2＋disp16 現在のコード・セグメント内の呼び出し元プログラムにリターンし，スタック・ポインタをdisp16だけ調整する．
RET	disp16	CA kk jj	3	23										〔PC〕←〔〔SP〕〕，〔SP〕←〔SP〕＋2， 〔CS〕←〔〔SP〕〕，〔SP〕←〔SP〕＋2＋disp16 他のコード・セグメントの呼び出し元プログラムにリターンし，スタック・ポインタをdisp16だけ調整する．

表 4-9　プログラム・カウンタ制御命令(続き)

ニーモニック	オペランド	オブジェクト・コード	バイト	クロック	ステータス									動作内容
					O	D	I	T	S	Z	A	P	C	
JMP	addr	EA kk jj hh gg	5	15										〔PC〕←addr(オフセット部),〔CS〕←addr(セグメント部) 他のコード・セグメントにジャンプする．新しいオフセット・アドレスはjjkkに，新しいセグメント・アドレスはgghhになる．
JMP	disp	EB kk	2	15										〔PC〕←〔PC〕＋disp プログラム・カウンタ相対のジャンプを行なう．
JMP	disp16	E9 kk jj	3	15										〔PC〕←〔PC〕＋disp16 現在のコード・セグメント内でジャンプを行なう．
JMP	mem (SEG ＋PC)	FF mod 101 r/m (DISP) (DISP)	2，3または4	24 ＋ EA										〔PC〕←〔mem〕，〔CS〕←〔mem＋2〕 他のコード・セグメント内へジャンプする．memで示されるメモリ位置の内容をPCに移動し，続くメモリ位置の内容をCSレジスタに移動する．
JMP	mem/reg (PCのみ)	FF mod 100 r/m (DISP) (DISP)	2，3または4	16 ＋ EA (mem) 11 (reg)										〔PC〕←〔mem/reg〕 現在のコード・セグメント内のメモリ位置にジャンプする．mem/regで示されるメモリ位置あるいはレジスタの内容をPCに移動する．

る．これにより，サブプログラムが操作するスタック上のパラメータの受け渡しができるように，RETURN命令がスタック・ポインタを調整することが可能になる．8086には，次の4つのRETURN命令がある．

	PCへのポップ	CSとPCへのポップ
通常のリターン	RET	RET
スタックへの ディスプレイスメントの加算	RET disp16	RET disp16

8086は，条件付きコールや条件付きリターンの命令を持っていないことに注意．8080に対応する命令を実現するには，CALLあるいはRETURNの命令を条件付きジャンプ命令と共に用いる必要がある．たとえば，次の8080の命令は，

```
                      CNZ     SUB$PROGRAM      CALL SUB$PROGRAM if ZF = 0
NEXT$INSTRUCTION:     ORA     A
```

以下の8086の命令で置き換えられる．

```
                      JZ      NEXT$INSTRUCTION   ;JUMP TO NEXT$INSTRUCTION if ZF =1
                      CALL    SUB$PROGRAM        ;JUMP TO SUB$PROGRAM if ZF = 0
NEXT$INSTRUCTION:     OR      AX,BX
```

8086のジャンプ命令は表4-9に示されている．8086のジャンプ命令は一般に，8086のＣＡＬＬ命令と同様の変形を有する．さらにジャンプ命令には，

```
JMP   disp
```

の形式があり，これは，3バイトのＪＭＰ disp16 の命令に対して，2バイトのオブジェクト・コードを持つ．ＪＭＰ disp は相対分岐で，プログラム・カウンタに8ビット符号付き2進数のディスプレイスメントを加算する．これにより，1ないし127バイト以内のジャンプ命令が可能になる．この章の多くのプログラム例では，ジャンプ命令を用いてその使用法を示している．

4.5.1 条件付きジャンプ命令

いろいろな条件に基づいてプログラム・カウンタの内容を変更する，8086の命令を**表4-10**に示す．

表4-11には，一般に用いられる算術演算の比較をあげ，8086でそれがどのように得られるかを示してある．

一般に，大きいとか小さいとかいうのは符号付き操作に用いられる形容詞であり，上（上位）あるいは下（下位）は符号なし操作に用いられる形容詞である．

ＣＸレジスタの内容を減少して，必要に応じてプログラム・カウンタの内容を変更する8086の命令を**表4-12**に示す．これらの命令は，一般にＬＯＯＰ命令と呼ばれる．練習として，この章の前節を検討して，

```
DEC    CX
JNZ    label
```

の命令構成を，

```
LOOP    label
```

で置き換えてみよ．

1つの置き換えは，1バイトのオブジェクト・コードの節約になる．さらに，1回の実行につき，1クロックが節約される．100回の繰り返しを行なうループの一度の実行について，これは，100クロックあるいは5MHzの8086で20マイクロ秒の節約を意味している．

JCXZの命令は，このグループの命令の中では特異な存在で，フラグ・レジスタの内容に基づいて分岐するのではなくて，CXレジスタが0ならばジャンプする．JCXZ命令は，LOOP命令と共にCXレジスタと関係を持つので，LOOP命令と共に表4-12にも示してある．

(1) LOOP命令

LOOP命令は，DEC　CXとJNZの命令を兼ねている．たとえば，図4-1の命令は，次のように書き改めることができる．

```
MOVE$BYTES:     MOV     AL, [SI]
                MOV     [DI], AL
                INC     SI
                INC     DI
                LOOP    MOVE$BYTES
```

以後，すべての命令において，LOOP命令は，

```
DEC     CX
JNZ     label
```

の命令で置き換えられる．

4.6　プロセッサ制御命令

フラグに作用し，8086の外部インターフェイスをいろいろな面から制御する，8086の命令を表4-13に示す．

表4-10　条件付き分岐命令

ニーモニック	オペランド	オブジェクト・コード	バイト	クロック	ステータス O	D	I	T	S	Z	A	P	C	動作内容
JA	disp	77 kk	2	4/16										(〔CF〕OR〔ZF〕)＝0のとき，〔PC〕←〔PC〕＋disp キャリーとゼロのフラグが0ならば分岐する．
JNBE	disp	JAと同じ												
JAE	disp	73 kk	2	4/16										〔CF〕＝0のとき，〔PC〕←〔PC〕＋disp キャリー・フラグが0ならば分岐する．
JNC	disp	JAEと同じ												
JNB	disp	JAEと同じ												
JB	disp	72 kk	2	4/16										〔CF〕＝1のとき，〔PC〕←〔PC〕＋disp キャリー・フラグが1ならば分岐する．
JC	disp	JBと同じ												
JNAE	disp	JBと同じ												
JBE	disp	76 kk	2	4/16										(〔CF〕OR〔ZF〕)＝1のとき，〔PC〕←〔PC〕＋disp キャリー・フラグあるいはゼロ・フラグが1ならば分岐する．
JNA	disp	JBEと同じ												
JE	disp	74 kk	2	4/16										〔ZF〕＝1のとき，〔PC〕←〔PC〕＋disp ゼロ・フラグが1ならば分岐する．
JZ	disp	JEと同じ												
JG	disp	7F kk	2	4/16										(〔ZF〕＝0 AND〔SF〕＝〔OF〕)＝1のとき，〔PC〕←〔PC〕＋disp ゼロ・フラグが0でサインとオーバーフローのフラグが等しければ分岐する．
JNLE	disp	JGと同じ												
JGE	disp	7D kk	2	4/16										〔SF〕＝〔OF〕のとき，〔PC〕←〔PC〕＋disp サインとオーバーフローのフラグが等しければ分岐する．
JNL	disp	JGEと同じ												
JL	disp	7C kk	2	4/16										〔SF〕≠〔OF〕のとき，〔PC〕←〔PC〕＋disp サインとオーバーフローのフラグが等しくなければ分岐する．
JNGE	disp	JLと同じ												
JLE	disp	7E kk	2	4/16										(〔SF〕＝〔OF〕AND〔ZF〕＝0)＝1のとき，〔PC〕←〔PC〕＋disp サインとオーバーフローのフラグが等しくてゼロ・フラグが0ならば分岐する．
JNG		JLEと同じ												

表4-10　条件付き分岐命令(続き)

ニーモニック	オペランド	オブジェクト・コード	バイト	クロック	ステータス									動作内容
					O	D	I	T	S	Z	A	P	C	
JNE	disp	75 kk	2	4/16										〔ZF〕＝0のとき,〔PC〕←〔PC〕＋disp ゼロ・フラグが0ならば分岐する.
JNZ	disp	JNEと同じ												
JNO	disp	71 kk	2	4/16										〔OF〕＝0のとき,〔PC〕←〔PC〕＋disp オーバーフロー・フラグが0ならば分岐する.
JNP	disp	7B kk	2	4/16										〔PF〕＝0のとき,〔PC〕←〔PC〕＋disp パリティ・フラグが0ならば分岐する.
JPO	disp	JNPと同じ												
JNS	disp	79 kk	2	4/16										〔SF〕＝0のとき,〔PC〕←〔PC〕＋disp サイン・フラグが0ならば分岐する.
JO	disp	70 kk	2	4/16										〔OF〕＝1のとき,〔PC〕←〔PC〕＋disp オーバーフロー・フラグが1ならば分岐する.
JP	disp	7A kk	2	4/16										〔PF〕＝1のとき,〔PC〕←〔PC〕＋disp パリティ・フラグが1ならば分岐する.
JPE	disp	JPと同じ												
JS	disp	78 kk	2	4/16										〔SF〕＝1のとき,〔PC〕←〔PC〕＋disp サイン・フラグが1ならば分岐する.
JCXZ	disp	E3 kk	2	6/18										〔CX〕＝0のとき,〔PC〕←〔PC〕＋disp CXレジスタが0ならば分岐する.

表4-11　符号付きと符号なしの比較命令

	符合付き		符号なし	
=	.EQ. JE または JZ	等しい、または0	JEまたはJZ	等しい、または0
≠	.NE. JNE またはJNZ	等しくない、または0でない	JNEまたはJNZ	等しくない、または0でない
>	.GT. JG またはJNLE	大きい、または小さくもなく等しくもない	JAまたはJNBE	上、または下でもなく等しくもない
≥	.GE. JGE またはJNL	大きいか等しい、または小さくはない	JAEまたはJNB	上か等しい、または下ではない
<	.LT. JL またはJNGE	小さい、または大きくもなく等しくもない	JBまたはJNAE	下、または上でもなく等しくもない
≤	.LE. JLE またはJNG	小さいか等しい、または大きくはない	JBEまたはJNA	下か等しい、または上ではない

表4-12　ループ命令

ニーモニック	オペランド	オブジェクト・コード	バイト	クロック	ステータス									動作内容
					O	D	I	T	S	Z	A	P	C	
LOOP	disp	E2 kk	2	5/17										〔CX〕←〔CX〕−1，〔CX〕≠0のとき，〔PC〕←〔PC〕＋disp CXレジスタを減少させるが，フラグには影響しない．CXレジスタが0でなければ分岐する．
LOOPE	disp	E1 kk	2	6/18										〔CX〕←〔CX〕−1，〔CX〕≠0で〔ZF〕＝1のとき，〔PC〕←〔PC〕＋disp CXレジスタを減少させるが，フラグには影響しない．CXレジスタが0でなく，しかもゼロ・フラグが1ならば分岐する．
LOOPZ	disp	LOOPEと同じ												
LOOPNE	disp	E0 kk	2	5/19										〔CX〕←〔CX〕−1，〔CX〕≠0で〔ZF〕＝0のとき，〔PC〕←〔PC〕＋disp CXレジスタを減少させるが，フラグには影響しない．CXレジスタが0でなく，しかもゼロ・フラグが0ならば分岐する．
LOOPNZ	disp	LOOPNEと同じ												
JCXZ	disp	E3 kk	2	6/18										〔CX〕＝0のとき，〔PC〕←〔PC〕＋disp CXレジスタが0ならば分岐する．

表4-13　プロセッサ制御命令

ニーモニック	オペランド	オブジェクト・コード	バイト	クロック	ステータス O	D	I	T	S	Z	A	P	C	動作内容
CLC		F8	1	2									0	〔CF〕←0 キャリー・フラグをクリヤする．
CMC		F5	1	2									X	〔CF〕←〔$\overline{CF}$〕 キャリー・フラグの補数をとる．
CLD		FC	1	2		0								〔DF〕←0 ディレクション・フラグをクリヤする．
CLI		FA	1	2			0							〔IF〕←0 インタラプト・フラグをクリヤして，すべてのインタラプトを無効にする．
STC		F9	1	2									1	〔CF〕←1 キャリー・フラグをセットする．
STD		FD	1	2		1								〔DF〕←1 ディレクション・フラグをセットする．
STI		FB	1	2			1							〔IF〕←1 インタラプト・フラグを1にして，インタラプトを有効にする．
NOP		90	1	3										無操作.
ESC	mem	11011xxx mod xxx r/m (DISP) (DISP)	2 3または4	8 + EA										memで示されるメモリ位置の内容を，アドレス/データ・バスに設定する．mod＝11のときは無操作となる．
LOCK		F0	1	2										後続の命令の実行中，バスの制御を保証する．
WAIT		9B	1	3または それ以上										外部ロジックが$\overline{TEST}$ピンをローにするまで，ウエート状態となる．
HLT		F4	1	2または それ以上										ホルト状態になる．

4.7 入出力命令

8086の入力と出力の機能を行なう命令を，**表4-14**に示す．

8086でのメモリ・マップド・アドレッシングは，I/Oポートのアドレス指定に重要な利点を持つ．メモリ・マップドI/Oアドレスのコード化がどのように行なわれるかによって，ストリング・プリミティブ命令はメモリに対して行なわれる繰り返し操作を可能にする．I/O ポート・アドレッシングが用いられている**図4-32**と，メモリ・マップド・アドレッシングが用いられている**図4-33**を比較せよ．この両方のルーチンはブロックのデータを出力する．ブロック中のバイト数はCXレジスタに含まれている．図4-32のルーチンは，DIレジスタで示されるブロックをIO$PORT に出力する．図4-33のルーチンは，SIレジスタで示されるブロックをDIレジスタで示されるメモリ・マップドI/Oポートに移動する．

I/Oポートのアドレスは，直接に指定されるか，あるいはDXレジスタに保持される．8ビットのアドレスは直接に指定され，16ビットのI/Oポート・アドレスはDXレジスタによって指定される．

```
OUTPUT$A$BYTE:    LODSB
                  OUT     IO$PORT,AL
                  LOOP    OUTPUT$A$BYTE
                  RET
```

図4-32 I/Oポート・アドレス指定によるブロックI/O

```
REP  MOVS
     RET
```

図4-33 メモリ・マップ・アドレス指定によるブロックI/O

MOVSはSIとDI中のアドレスを自動的に増加あるいは減少させるので，ブロックのメモリ・アドレスはメモリ・マップドI/Oポートに割り付けられる必要のあることに注意．

表4-14　8086の I/O 命令

ニーモニック	オペランド	オブジェクト・コード	バイト	クロック	ステータス									動作内容
					O	D	I	T	S	Z	A	P	C	
IN	ac,DX	1110110w	1	8										〔ac〕←〔PORTDX〕 DXレジスタで示されるI/Oポートから,AL(8ビット操作)あるいはAX(16ビット操作)のレジスタに入力する.
IN	ac,port	1110010w kk	2	10										〔ac〕←〔port〕 命令の２番目のバイトで示されるI/Oポートから,AL(8ビット操作) あるいはAX (16ビット操作) のレジスタに入力する.
OUT	ac,DX	1110011w	1	8										〔PORTDX〕←〔ac〕 DXレジスタで示されるI/Oポートに，AL(8ビット操作) あるいはAX (16ビット操作) のレジスタの内容を出力する.
OUT	ac,port	1110111w kk	2	10										〔port〕←〔ac〕 命令の２番目のバイトで示されるI/Oポートに，AL (8ビット操作) あるいはAX (16ビット操作) のレジスタの内容を出力する.

4.8　インタラプト命令

ソフトウエア・インタラプト命令，オーバーフローに関するインタラプト命令，インタラプトから復帰する命令を**表4-15**に示す．

ソフトウエア・インタラプト命令は，次の主要な目的に用いられる．

1．プログラムのデバッグ．1バイトのソフトウエア・インタラプト命令は，そのアドレスが$0000C_{16}$にあるルーチンを呼び出す．通常，このルーチンはデバッグ・パッケージの一部であり，ブレークポイントの処理に用いられる．
2．そのアドレスがメモリの最初の1024バイトに存在するサブルーチンの呼び出し．2バイトのソフトウエア・インタラプト命令が用いられると，そのアドレスがメモリの最初の1024バイトにある256個のサブルーチンの1つが呼び出される．

ソフトウエア・インタラプト命令は，セグメント間ＣＡＬＬで用いられるプログラム・メモリの5バイトと比較して，1あるいは2バイトのプログラム・メモリを用いる利点を持っている．さらに，ソフトウエア・インタラプトは，フラグ・レジスタをスタックに自動的にセーブする．これは多くの場合，望ましい特徴である．小さな欠点は，ソフトウエア・インタラプトによってルーチンが呼ばれると，このルーチンからはＩＲＥＴ命令によって復帰する必要があり，これはＲＥＴ命令より多くの時間を要することがあげられる．

表4-15　8086のインタラプト命令

ニーモニック	オペランド	オブジェクト・コード	バイト	クロック	ステータス									動作内容
					O	D	I	T	S	Z	A	P	C	
INT		1100110v kk(v=1のとき)					0	0						〔SP〕←〔SP〕−2,〔〔SP〕〕←〔FLAGS〕,〔IF〕←0,〔TF〕←0, 〔SP〕←〔SP〕−2,〔〔SP〕〕←〔CS〕,〔SP〕←〔SP〕−2, 〔〔SP〕〕←〔PC〕,〔CS〕←〔ベクタ(セグメント部)〕, 〔PC〕←〔ベクタ(オフセット部)〕
		v=0	1	52										v=0のとき,ベクタ(オフセット部)=〔0000C16〕 ベクタ(セグメント部)=〔0000E16〕
		v=1	2	51										v=1のとき,ベクタ(オフセット部)=〔(kk*4)〕 ベクタ(セグメント部)=〔(kk*4)+2〕 ソフトウエア・インタラプトを発生させる.
INTO		CE	1	53 (OF=1) 4 (OF=0)			0	0						〔OF〕=1のとき,〔SP〕←〔SP〕−2,〔〔SP〕〕←〔FLAGS〕,〔IF〕←0, 〔TF〕←0,〔SP〕←〔SP〕−2,〔〔SP〕〕←〔CS〕,〔SP〕←〔SP〕−2, 〔〔SP〕〕←〔PC〕,〔CS〕←〔00012_{16}〕,〔PC〕←〔00010_{16}〕. オーバーフロー・フラグが1ならば，オーバフロー処理のベクタによってソフトウエア・インタラプトを発生させる．そうでなければ，次に続く命令を実行する．
IRET		CF	1	32	X	X	X	X	X	X	X	X	X	〔PC〕←〔〔SP〕〕,〔SP〕←〔SP〕+2,〔CS〕←〔〔SP〕〕,〔SP〕←〔SP〕+2,〔FLAGS〕←〔〔SP〕〕,〔SP〕←〔SP〕+2 インタラプト・サービス・ルーチンからリターンする．

4.9 ローテートとシフトの命令

ローテートとシフトを行なう，8086の命令を表4-16に示す．

ローテートとシフトの命令は，ビットを調べる操作に多く用いられる．これらの命令は，種々のビット・パターンを調べるために，単独で，あるいは論理演算操作と共に用いられる．レジスタの最下位ビットを調べるには，

```
ROR   reg, 1
```

の命令の方が，

```
AND   reg, 01H
```

あるいは，

```
TEST   reg, 01H
```

の命令よりも１サイクル速い．ゼロ・フラグが意味を持つＡＮＤあるいはＴＥＳＴの命令に対して，ＲＯＲ命令はキャリー・フラグを調べる．16ビットのポインタやレジスタの最下位ビットを調べるためには，

```
ROR   reg, 1
```

を用いる．この命令ではオブジェクト・コードが１バイト節約され，処理は，

```
AND   reg, 0001H
```

あるいは，

```
TEST   reg, 0001H
```

の命令よりも１サイクル速い．ただし，ＲＯＲ命令はレジスタの内容を変えるのに対し，ＴＥＳＴ命令は非破壊であることに注意．

上に述べたように，８ビット・レジスタあるいは16ビット・レジスタの最上位ビットは，ＲＯＲ命令をＲＯＬ命令で置き換えることによって調べられる．

ローテートとシフトは算術演算操作を行なう．算術的なシフト操作は，乗算と除算を行なうために用いることができる．図4-34のルーチンは，２桁のＡＳＣＩＩキャラクタを16進数に変換する．このルーチンは，ＳＩレジスタが２つのキャラクタ（上位バイトが最初に位置する）を示し，結果をＡＬレジスタに入れて復帰することを仮定している．またこのルーチンでは，変換されるバイトが，０から９あるいはＡからＦの範囲にあることを保証している．もしこの範囲外のキャラクタであれば復帰時にゼロ・フラグを１にし，そうでなければゼロ・フラグを０にする．

```
CONVERT$TWO$ASCII$TO$HEX    PROC    NEAR

                            PUSH    CX
                            LODSB                           ;LOAD FROM SI TO AL
                            CALL    CONVERT$ASCII$TO$HEX
                            JZ      TRANSLATION$ERROR

                            MOV     CL,4                    ;SET UP FOR ROTATE
                            SAL     AL,CL                   ;ROTATE FOUR TIMES
                            MOV     AH,AL                   ;SAVE IN AH
                            LODSB
                            CALL    CONVERT$ASCII$TO$HEX
                            JZ      TRANSLATION$ERROR

                            OR      AL,AH                   ;CREATE THE HEX BYTE
                            OR      AH,0FFH                 ;TURN ZF=0
                            POP     CX
                            RET

TRANSLATION$ERROR:          RET                             ;ZF IS KNOWN TO BE 1

CONVERT$TWO$ASCII$TO$HEX    ENDP

CONVERT$ASCII$TO$HEX        PROC    NEAR

                            SUB     AL,30H
                            JL      TRANNY$ERROR
                            CMP     AL;0AH                  ;IS IT 0 - 9
                            JL      DONE
                            SUB     AL,07H                  ;ADJUST FOR A - F.
                            CMP     AL,10H                  ;IS IT MORE?
                            JGE     TRANNY$ERROR
DONE:                       RET

TRANNY$ERROR:               XOR     AH,AH
                            RET

CONVERT$ASCII$TO$HEX        ENDP
```

図4-34　ASCII表示の2桁を16進数に変換するルーチン

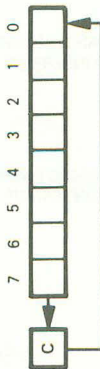
表4-16　8086のシフトとローテートの命令

ニーモニック	オペランド	オブジェクト・コード	バイト	クロック	ステータス									動作内容
					O	D'	I	T	S	Z	A	P	C	
RCL	mem/reg, count	110100vw mod 010 r/m (DISP) (DISP)	2, 3または4	count = 1, reg: 2 mem: 15 + EA count = [CL] reg: 8 + 4*[CL] mem: 20 + EA + 4 *[CL]	X								X	mem/regで示されるメモリ位置あるいはレジスタの内容を，キャリー・フラグと共に左へローテートする．ローテートのビット数はcountで決まり，1(v＝0) あるいはCLレジスタの内容（v＝1）となる．ローテートは次のように行なわれる． w＝0のとき C ← 7 6 5 4 3 2 1 0 w＝1のとき C ← 15 14 13 … 2 1 0

表4-16　8086のシフトとローテートの命令(続き)

ニーモニック	オペランド	オブジェクト・コード	バイト	クロック	ステータス									動作内容
					O	D	I	T	S	Z	A	P	C	
RCR	mem/reg, count	110100vw mod 011 r/m (DISP) (DISP	2, 3または4	count = 1, reg: 2 mem: 15 + EA count = [CL] reg: 8 + 4*[CL] mem: 20 + EA + 4* [CL]	X								X	mem/regで示されるメモリ位置あるいはレジスタの内容を，キャリー・フラグと共に右へローテートする．ローテートのビット数はcountで決まり，1(v=0) あるいはCLレジスタの内容（v=1）となる．ローテートは次のように行なわれる． w=0のとき 7 6 5 4 3 2 1 0 C w=1のとき 15 14 13 2 1 0 C

表4-16　8086のシフトとローテートの命令(続き)

ニーモニック	オペランド	オブジェクト・コード	バイト	クロック	ステータス O	D	I	T	S	Z	A	P	C	動作内容
ROL	mem/reg, count	110100vw mod 000 r/m (DISP) (DISP)	2, 3または4	count = 1, reg: 2 mem: 15 + EA count = [CL] reg: 8 + 4*[CL] mem: 20 + EA + 4* [CL]	X								X	mem/regで示されるメモリ位置あるいはレジスタの内容を左へローテートする．オペランドの最上位ビットはキャリー・フラグにはいる．ローテートのビット数はcountで決まり，1(v = 0)あるいはCLレジスタの内容(v = 1)となる．ローテートは次のように行なわれる． w = 0のとき w = 1のとき

表4-16　8086のシフトとローテートの命令(続き)

ニーモニック	オペランド	オブジェクト・コード	バイト	クロック	ステータス									動作内容
					O	D	I	T	S	Z	A	P	C	
ROR	mem/reg, count	110100vw mod 001 r/m (DISP) (DISP)	2, 3または4	count = 1, reg: 2 mem: 15 + EA count = [CL] reg: 8 + 4*[CL] mem: 20 + EA + 4*[CL]	X								X	mem/regで示されるメモリ位置あるいはレジスタの内容を右へローテートする．オペランドの最下位のビットはキャリー・フラグにはいる．ローテートのビット数はcountで決まり，1(v＝0)あるいはCLレジスタの内容(v＝1)となる．ローテートは次のように行なわれる． w＝0のとき 7 6 5 4 3 2 1 0 / C w＝1のとき 15 14 13 2 1 0 / C

表4-16　8086のシフトとローテートの命令(続き)

ニーモニック	オペランド	オブジェクト・コード	バイト	クロック	ステータス									動作内容
					O	D	I	T	S	Z	A	P	C	
SAL	mem/reg, count	110100vw mod 100 r/m (DISP) (DISP)	2, 3または4	count = 1, reg: 2 mem: 15 + EA count = [CL] reg: 8 + 4*[CL] mem: 20 + EA + 4*[CL]	X				X	X	?	X	X	mem/regで示されるメモリ位置あるいはレジスタの内容を左へシフトする．オペランドの最下位ビットには0がはいる．シフトのビット数はcountで決まり，1 (v=0) あるいはCLレジスタの内容 (v=1) となる．シフトは次のように行なわれる． w=0のとき w=1のとき
SHL	mem/reg, count	SALと同じ												

表4-16　8086のシフトとローテートの命令(続き)

ニーモニック	オペランド	オブジェクト・コード	バイト	クロック	ステータス									動作内容
					O	D	I	T	S	Z	A	P	C	
SAR	mem/reg, count	110100vw mod 111 r/m (DISP) (DISP)	2, 3または4	count = 1, reg: 2 mem: 15 + EA count = [CL] reg: 8 + 4*[CL] mem: 20 + EA + 4*[CL]	X				X	X	?	X	X	mem/regで示されるメモリ位置あるいはレジスタの内容を，右へシフトする．最上位ビットの値を残すために，オペランドの符号を拡張する．シフトのビット数はcountで決まり，1(v＝0) あるいはCLレジスタの内容 (v=1) となる．シフトは次のように行なわれる． w＝0のとき C　6 5 4 3 2 1 0 w＝1のとき C　14 13 … 2 1 0

表4-16　8086のシフトとローテートの命令(続き)

ニーモニック	オペランド	オブジェクト・コード	バイト	クロック	ステータス									動作内容
					O	D	I	T	S	Z	A	P	C	
SHR	mem/reg, count	111100vw mod 101 r/m (DISP) (DISP)	2, 3または4	count = 1, reg: 2 mem: 15 + EA count = [CL] reg: 8 + 4*[CL] mem: 20 + EA + 4 * [CL]	X				X	X	?	X	X	mem/regで示されるメモリ位置あるいはレジスタの内容を，右へシフトする．オペランドの最上位からは0を入れる．シフトのビット数はcountで決まり，1(v=0) あるいはCLレジスタの内容 (v=1) となる．シフトは次のように行なわれる． w=0のとき C　0→ 7 6 5 4 3 2 1 0 w=1のとき C　0→ 15 14 13 … 2 1 0

第5章
ソフトウエア開発

ソフトウエア開発過程での大きな補助となる，次の3つの道具がある．

●エディタ

エディタは，ソース・コードの入力や修正に用いられる．ソース・コードは普通，フロッピー・ディスク，ハード・ディスク，または非常の場合にペーパ・テープなどの，ある形式のマス記憶にセーブされる．ソース・コードは通常，マス記憶ではソース・ファイルと呼ばれるファイルの形式で構成されている．エディタは，ソース・ファイルのアクセス方法に依存して，いくつかの方法でソース・ファイルの作成と処理を行なう．たとえば，マグネティック・テープ中のソース・ファイルは，フロッピー・ディスク中のソース・ファイルと同じ方法では処理されない．ユーザは一般に，ビデオ・ターミナルからコマンドを入力して，エディタの機能を行なわせる．

●アセンブラ

アセンブラは，ソース・コードをオブジェクト・コードに翻訳するために用いられる．アセンブラは，ほとんどの場合，エディタによって作られたソース・ファイルを読み出して，ソース・コードを翻訳し，オブジェクト・ファイルと呼ばれるファイルの形式で，マス記憶にオブジェクト・コードを書き込む．アセンブラはまた，リスティング・ファイルと呼ばれる，ソース・コードとオブジェクト・コードを含むファイルや，シンボル・ファイルと呼ばれる，ソース・コードで用いられているすべてのラベルや変数名を含むファイルなど，付加的なものも作成する．リスティング・ファイルは普通，印刷されて，デバッグの過程で参照される．

●デバッガ

デバッガは，オブジェクト・コード中のエラー検出を援助するために用いられる．オブジェクト・ファイルがマス記憶からロードされることを要求できる位置に，デバッガはアセンブラによって生成されたオブジェクト・コードとロードされるか，あるいは単独にロードされる．代表的なデバッガでは，オブジェクト・コードの実行を制御し，メモリとレ

ジスタの内容を見ることができる．

以上述べた3つのプログラムは，開発過程に本質的で主要な道具である．その他の有用な道具には，リンカとローダがある．リンカは，複数のサブプログラムを1つのプログラムに結合するために用いられる．リンカは，サブプログラムからの外部参照を解決する．外部参照は，1つのモジュール内の命令が，他のモジュールで定義されているシンボル(ラベルまたは変数名）を参照するときに生じる．ローダは，マス記憶からメモリにオブジェクト・コードを移すために用いられる．

これを論じる目的で，ソフトウエア開発過程で用いる仮想システムを考える．これには，次のハードウエア要素が含まれている．

- ＣＰＵ
- ＲＡＭ
- フロッピー・ディスク装置
- ＣＲＴターミナル
- プリンタ

これらの要素の接続を**図5-1**に示す．

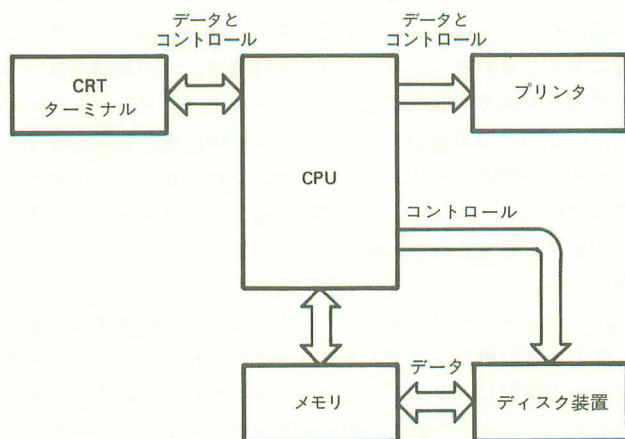



図5-1　仮想の開発システム

エディタ，アセンブラ，そしてデバッガの解説を通して，これら支援プログラムは上記のシステムで実行されることを仮定している．さらに，各支援プログラムの解説に，支援プログラムが備えている代表的な機能を示す例で，支援プログラムの仮想的コマンド言語を用いている．このシステムとコマンド言語は単なる例であって，実在しないことを強調しておく．

5.1　エディタ

多くのエディタは，次の作業を組み合わせることによってその機能を果たしている．

- マス記憶からメモリにデータを読み込む．
- ユーザのコマンドに応えて，メモリ中のデータを処理する．
- メモリからマス記憶にデータを書き出す．

この操作の例を**図5-2**に示す．

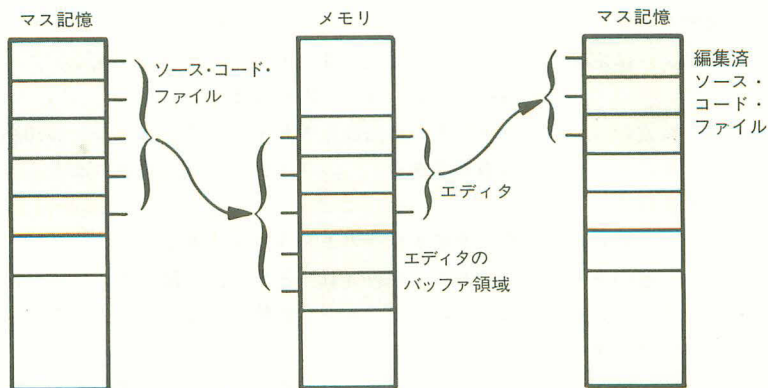



図5-2　基本的なエディタの操作

次の用語は一般に，基本的なタイプのエディタを述べる際に用いられる．

バッファ：データはマス記憶から，バッファと呼ばれるメモリ領域に読み込まれる．すべての編集コマンドは，バッファ内のデータを操作する．データは処理が終わると，バッファからマス記憶に書かれる．

キャラクタまたはラインのポインタ：エディタは，バッファ内にポインタを保持している．ユーザのコマンドはすべてこのポインタに相対とみなされる．たとえば，4つのキャラクタを削除するコマンドは，ポインタの後の4つのキャラクタを削除する．バッファ内の特定のキャラクタを参照するポインタを用いるエディタもある．このようなエディタは，キャラクタ指向のエディタと呼ばれる．特定の行を参照するポインタを用いるエディタもあり，これはライン指向のエディタと呼ばれる．

5.1.1　エディタの機能

エディタはどのような種類の機能を備えるべきであろうか．

エディタは次のような能力を備えている必要がある．

● マス記憶からメモリへのデータのリード
● メモリからマス記憶へのライト
● メモリへのデータの挿入
● メモリからのデータの削除
● バッファ内のキャラクタまたはラインのポインタ位置の変更
● バッファ内容の表示
● 指定ストリングを有するバッファの検索
● バッファ内容の変更
● システム・コマンド

これらの機能を示すために用いるサンプルのエディタは，CRTターミナルで入力されるユーザのコマンドに応答する．コマンドは次の3つのフィールドから成る．

ナンバー　　コマンド　　ストリング

ナンバーは，特定のコマンドが実行される回数を表わす．このフィールドは10進数として解釈される．このフィールドは省略できる．このフィールドが省略されると，デフォルト値に1が仮定される．

コマンドは，行なわれるべき操作を示す単一のキャラクタである．

ストリングは一連のキャラクタである．実行されるときに，1個あるいはそれ以上のストリングを用いるコマンドがある．このフィールドは省略できる．ストリングは，“#”のキャラクタあるいはリターンのキャラクタで終わる．

すべてのコマンドは，キャリッジ・リターンで終わる．これをⓡで表わす．次はコマンドの例である．

Aⓡ	バッファに1行追加
10Lⓡ	バッファ内のポインタを10行下に移動
CTHE#ANⓡ	ストリングの THE を AN に変更

(1) メモリに対するデータの読み込みと書き込み

エディタは，バッファからマス記憶に読み込みあるいは書き込みを行なう能力を備えている必要がある．ユーザは，転送されるデータの量が指定できなければならない．代表的なデータの量には次のものがある．

● 1個または複数のキャラクタ：
● 1行または複数行：たとえば，1行の転送は，キャリッジ・リターンが検出されるまで，すべてのデータを移動する．n行の転送は，n個のキャリッジ・リターンが検出されるまで，すべてのデータを移動する．
● バッファ全体：読み込み操作に対しては，バッファが満たされるまで，マス記憶からバッファへのデータ移動を含む．書き込み操作に対しては，バッファ全体がマス記憶に移動されるまで，バッファからマス記憶へのデータ移動を含む．

有用となる付加的特徴には次のものがある．

● マス記憶からデータを転送して，転送したデータを削除する操作：
● 特定のキャラクタが検出されるまで，データを転送する読み込みあるいは書き込みの

操作：たとえば，ページ単位のデータ転送が可能となる．すなわち，ページ最後のキャラクタ（フォーム・フィード）が検出されるまで，すべての情報を転送する．

例として，バッファに行を付加するサンプルのエディタ・コマンドがAである場合を考える．コマンド

A(r)

はバッファに1行付加する．コマンド

10A(r)

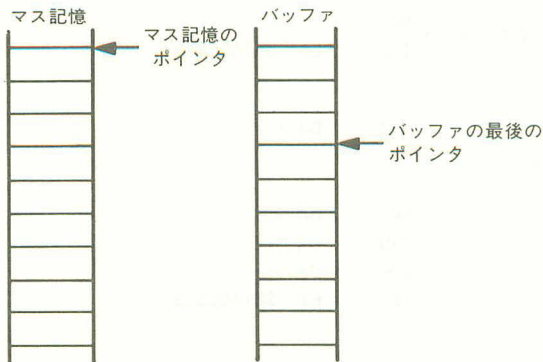
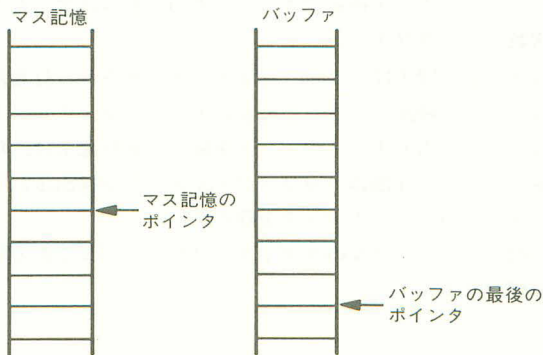
はバッファに10行付加する．コマンド

!A(r)

はバッファをマス記憶からの情報で満たす．コマンド

5A

の実行は次のように図示される．

5Aの実行前

5Aの実行後

(2) バッファへのデータ挿入

エディタは，バッファにデータを付加できる能力を備えていなければならない．ユーザは一般に，次の2つのタイプのソース・コード挿入の1つを行なう必要がある．

- 大量のソース・コードを挿入しなければならない．これは，最初にソース・コードが入力されるとき，あるいはソース・コードの大きい変更を行なうときに生じる．
- 1行または2行のソース・コードを入力しなければならない．これは，デバッグの過程で“バグ”が訂正されるとき，あるいはソース・コードが最初に入力されて，不注意にも1行または2行の入力を忘れていたことが発見されたときに生じる．

多くのエディタは，2つの異なる挿入モードを備えて，これら2つの要求に応じている．すなわち，ユーザが無制限のデータを入力できるモードと，ユーザによって制限された量のデータが入力されるモードがある．

データ挿入のサンプルのエディタ・コマンドがIである場合を考える．次のようなバッファを仮定する．

```
              MOV    CX,AX
              ADD    DX,SP
ポインタ──→   JNC    EXIT$STAGE$LEFT
```

コマンド

```
        I     SHR    DX,1 ⓡ
```

が入力されると，バッファは

```
              MOV    CX,AX
              ADD    DX,SP
              SHR    DX,1
              JNC    EXIT$STAGE$LEFT
```

に変更される．

(3) バッファからのデータ削除

エディタは，バッファからデータを除く能力を備えていなければならない．ユーザは，除かれるべきソース・コードの量を指定できる．代表的な量には次のものがある．

- 1個あるいは複数のキャラクタ:
- 1行または複数行: 1行除去は，ライン・ポインタで示される行を除くか，あるいはキャリッジ・リターンが検出されるまで現在のキャラクタ・ポインタで示されるキャラクタで始まるバッファ内のすべてのデータを除く．n行除去は，現在のライン・ポインタ以降のn行，またはn個のキャリッジ・リターンが検出されるまで現在のキャラクタ・ポインタからのすべてのデータを除く．

バッファから行を削除するサンプルのエディタ・コマンドがKである場合を考える．次のようなバッファを仮定する．

```
ポインタ─→ MOV    CX,AX
           ADD    DX,SP
           SHR    DX,1
           JNC    EXIT$STAGE$LEFT
```

コマンド

2K

が入力されると，バッファは

```
MOV    CX,AX
JNC    EXIT$STAGE$LEFT
```

に変更される．

(4) **キャラクタまたはラインのポインタ移動**

エディタは，キャラクタまたはラインのポインタをバッファ内の異なる位置に移動する能力を備えていなければならない．ユーザは，ポインタが移動すべきキャラクタまたはラインの数を指定できる．さらに有用な次の能力を持っている．

- キャラクタまたはラインのポインタの，バッファの先頭への移動．
- キャラクタまたはラインのポインタの，バッファの最後への移動．
- キャラクタまたはラインのポインタの，バッファ内の特定の行への移動．たとえば，キャラクタまたはラインのポインタをバッファの行番号11へ移動することが要求できる．

キャラクタまたはラインのポインタをバッファ内で上下に移動するためのサンプルのエディタ・コマンドがLである場合を考える．次のようなバッファを仮定する．

```
ポインタ─→ MOV    CX,AX
           ADD    DX,SP
           SHR    DX,1
           JNC    EXIT$STAGE$LEFT
           TEST   BX,40H
           JZ     DONT$MESS$WITH$BILL
```

コマンド

4L

が入力されると，バッファは変化しないが,ポインタは JZ DONT$MESS$WITH$BILL の命令を示す．さらにコマンド

−3L

により，バッファはやはり変化しないが，ポインタは SHR DX,1 の命令を示す．

(5) **バッファ内容の表示**

エディタは，バッファをユーザのターミナルに表示する能力を備えていなければならない．ユーザは，表示されるキャラクタや行の数を指定することができる．さらに有用な能力に次のものがある．

- ＣＲＴターミナルがあれば，スクリーン全体にデータを自動的に表示するのに便利である．さらに，一度に１行あるいは画面全体に表示する，バッファによるスクロール

の能力もさらに便利である．

バッファから行をＣＲＴターミナルに表示するサンプルのエディタ・コマンドがＴである場合を考える．バッファ内容が

```
             IN     AL,TOUCH$TONE$DECODER$PORT
ポインタ──→ CMP    AL,COLUMN$4$DIGIT
             JNZ    TOUCH$TONE$ENCODE
             MOV    [DI],MESSAGE$STARTED$CODE
```

で，コマンド

2T

を入力すると，

```
CMP    AL,COLUMN$4$DIGIT
JNZ    TOUCH$TONE$ENCODE
```

の行がＣＲＴターミナルに表示される．

(6) ストリングのバッファ検索

エディタは，ユーザ指定のキャラクタのストリングのバッファ検索の能力を備えている必要がある．非常に有用な付加的能力に，特定のストリングを含むすべてのソース・コードの検索がある．たとえば，ソース・コードが最初に入力されたとき，入力間違いがしばしば起こる．検索機能を変更機能と共に用いることによって，ソース・コードを最小の労力で修正することができる．

バッファ検索のサンプルのエディタ・コマンドがＳである場合を考える．もし，バッファが

```
             IN     AL,TOUCH$TONE$DECODER$PORT
ポインタ──→ CMP    AL,COLUMN$4$DIGIT
             JNZ    TOUCH$TONE$ENCODE
             MOV    [DI],MESSAGES$STARTED$CODE
```

で，コマンド

STONEⓡ

を入力すると，その結果は，エディタでキャラクタ・ポインタあるいはライン・ポインタのどちらを採用しているかに依存する．ライン・ポインタの場合，バッファは変化しない．しかし，ライン・ポインタの位置は次のように変更される．

```
             IN     AL,TOUCH$TONE$DECODER$PORT
             CMP    AL,COLUMN$4$DIGIT
ポインタ──→ JNZ    TOUCH$TONE$ENCODE
             MOV    [DI],MESSAGE$STARTED$CODE
```

キャラクタ・ポインタの場合は，ポインタは次のように変更される．

```
             IN     AL,TOUCH$TONE$DECODER$PORT
ポインタ─────CMP    AL,COLUMN$4$DIGIT
             JNZ    TOUCH$→TONE$ENCODE
             MOV    [DI],MESSAGE$STARTED$CODE
```

(7) バッファのストリングの変更

エディタは，バッファ内のデータを変更する能力を備えていなければならない．ユーザは，バッファに存在する任意のストリングをユーザ指定のストリングで置き換えることを指定できる．削除機能は，ユーザ指定のストリングがない，変更機能の縮小された場合と考えることができる．

バッファのデータを変更するサンプルのエディタ・コマンドが　Cストリング1＃ストリング2　である場合を考える．ここでコマンドは，次にバッファ内でストリング1の位置する場所を探して，それをストリング2で置き換えて，機能を果たす．バッファが次のようであるとする．

```
                IN     AL, TOUCH$TONE$DECODER$PORT
ポインタ ──→   CMP    AL, COLUMN$4$DIGIT
                JNZ    TOUCH$TONE$ENCODE
                MOV    [DI], MESSAGE$STARTED$CODE
```

コマンド

CCODE＃TRANCEⓡ

を入力すると，バッファは次のように変更される．

```
                IN     AL, TOUCH$TONE$DECODER$PORT
                CMP    AL, COLUMN$4$DIGIT
ポインタ ──→   JNZ    TOUCH$TONE$ENTRANCE
                MOV    [DI], MESSAGE$STARTED$CODE
```

5.1.2　システム・コマンド

エディタは，ユーザが合理的な方法で編集を終了させられるコマンドを備えていなければならない．合理的終了方法には次のものがある．

- バッファ内のすべてのデータをマス記憶へ移動する．
- バッファを通してすべての未処理のソース・コードをマス記憶に移動する．これは標準的な終了方法と考えられる．
- バッファをクリヤせずに直ちに編集を終える．この方法は，分別が足らないか運の悪いユーザの操作がソース・コードに破滅的な影響を与えた場合に用いられる．マス記憶の形式によっては，ソース・コードを原形に復元することができる．

上のタイプのコマンドは，基本的なエディタにとって必要な要素である．より高度なエディタは次のような能力を持つ．

1．個々のコマンドを，コマンド・ストリングに連結する：
2．コマンド・ストリングの多重繰り返し：たとえば，これはソース・コード中の特定のストリングすべてを変更する場合に特に有用である．
3．より高度なファイル処理：この解説では，ファイルの概念は避けている．ソース・コードは，マス記憶に存在する点から論じているだけである．より進んだエディタは，

たとえばハード・ディスクなどの高速マス記憶から一般に起動され，有力なデータ・ファイル操作能力を備えているオペレーティング・システムとのインターフェイスを持つ．実際，バッファからの読み込みと書き込みのユーザの責任を軽減するエディタがある．このエディタでは，ソース・コードの読み込みや書き込みについて考えることなく，ソース・コード全体をスクロールさせることができる．この機能はエディタによって自動的に行なわれる．

4．算術演算能力：非常に有力な計算器として用いられるエディタもある．
5．バッファのある部分を抜き出して，後で用いるために保存する能力：この能力は，ソース・コードの再配列が必要なとき，たとえば，ファイルの先頭の100行のソース・コードをファイルの中央に移動しなければならないとき，非常に有用となる．
6．ストリング操作に“あいまいな”要素を含む能力：たとえば，第1のキャラクタがAで，最後の3つのキャラクタがCDEで，第2のキャラクタがあいまいな，すなわち任意のキャラクタの，5個のキャラクタのストリングを検索するために，検索操作にはA＊CDEを用いる．

5.2 アセンブラ

多くのアセンブラは次の機能を行なう．

- アセンブリ言語のソース・コードを別々のステートメントに分割する：
- アセンブリ言語のステートメントを構成部分に分ける：この部分には，ラベル，アセンブリ言語のオペレータ，アセンブラ命令，アセンブリ言語のオペレータのためのオペランド，さらに注釈が含まれる．
- アセンブリ言語の規則に従って，構成部分を処理する：この処理から，アセンブラはオブジェクト・コード・ファイルとシンボル・テーブルを生成する．
- ファイルをマス記憶に書き込む：このファイルには，オブジェクト・コード・ファイル，リスティング・ファイル（これはオブジェクト・コード・ファイルとソース・コード・ファイルから成る），さらにシンボル・テーブル・ファイルが含まれる．

ソース・コードをアセンブリ言語のステートメントに分割することは，ほとんどのソース・コード・ファイルは1行が1つのステートメントで構成されている，すなわち，2つのキャリッジ・リターンの間には1つのステートメントが存在するので，かなり容易な作業である．1行に1つ以上のステートメントが可能なアセンブラもある．このようなステートメントは通常，キャリッジ・リターンと同様に取り扱われる特殊な区切りキャラクタで分離されている．

アセンブラを非常に有用な道具にする機能は，アセンブリ言語のステートメントの処理にある．アセンブリ言語のステートメントは次のような部分を持つ．

- ラベル：アセンブリ言語の命令には，ラベルがある場合とない場合がある．ラベルが存在すれば，ロケーション・カウンタの現在の値と共にシンボル・テーブルに記憶される．より複雑なアセンブラでは，オペレータのタイプあるいは命令によって，より

多くの情報が記憶される．

- オペレータ：オペレータは，ＡＤＣ，ＳＴＤ，ＩＮなどのアセンブリ言語のニーモニックかあるいはアセンブラ命令である．アセンブラ言語のニーモニックは，アセンブラによってオブジェクト・コードに翻訳される．たとえば，ＡＤＣ命令は数100の異なるオブジェクト・コードを生成するが，ＡＤＣ ＡＸ，ＤＸ の命令は唯一のオブジェクト・コードを生成する．

アセンブラ命令は，アセンブラがオブジェクト・コード・ファイルとリスティング・ファイルを生成するときに用いる各種の機能を制御するために用いられる．これは次のような制御を行なう．

- ソース・コードがアセンブルされるロケーション：プログラム・メモリのロケーションに対して絶対アドレスを含むプログラムは，それがメモリのどこに位置するかを知る必要がある．サンプルのアセンブラのロケーション指定の命令が，ＯＲＧである場合を考える．アセンブラ命令

```
ORG 0400H
```

がソース・コードに含まれていれば，これに続くアセンブリ言語のステートメントは，プログラム・カウンタが0400Ｈに設定されているとしてアセンブルされる．

- プログラムの実行開始アドレス：これは通常，オブジェクト・コード内に記憶される．ほとんどのアセンブラでは，プログラムの開始アドレスを，ソース・コードの最後のステートメント，ＥＮＤステートメントで指定することができる．ソース・アセンブラが開始アドレスを指定するためにＥＮＤステートメントを用いていると仮定する．

```
END    START$OF$PROGRAM
```

のステートメントが，ソース・コードの最後のステートメントならば，アセンブラは，STARTOFPROGRAMのアドレスを開始アドレスとして含むオブジェクト・コードを生成する．

- リスティング・ファイルの形式：リスティング・ファイルの形式を制御する命令は，ページ数を付けたり，ソースとオブジェクトのコードのある部分がリスティング・ファイルに含まれるかどうかを決めたり，リスティング・ファイルに関連した見出しなどの制御を行なう．
- データ・メモリのロケーションの初期値：
- オペランド：オペランドを必要とするアセンブリ言語のニーモニックと命令に対して，オペランドは次のように種々の異なる形で用いられる．

　　レジスタ名
　　数値（いくつかの異なるベースの１つで）
　　変数名
　　ラベル
　　ＡＳＣＩＩキャラクタのストリング式（算術演算あるいは論理演算のオペレータによる上記の組み合せ）

- 注釈：注釈は，プログラムの動作の説明に用いられる．これは，アセンブラでは無視．

されるが，プログラムの修正に興味を持つユーザには必要である．

アセンブラが行なう翻訳過程は，かなり簡単な作業である．ステートメントが構成部分に分割されると，構成部分からテーブルを用いてオブジェクト・コードの部分を作る．これを次に最終的なオブジェクト・コードに組み立てる．

5.3 デバッガ

デバッガは，オブジェクト・プログラムからエラーを取り除く際の援助に用いられる開発の道具である．デバッガは，エディタとアセンブラと同様に，複雑に異なる．最も基本的なデバッガは，ユーザに次のことを可能とする要素を含む．

- 実行の制御
- レジスタまたはメモリの表示

デバッガでは，次のような機能を用いて実行を制御できる．

- シングル・ステップの機能：シングル・ステップの機能で，オブジェクト・コードを一度に1命令実行することができる．ユーザは各命令実行の間のレジスタやメモリを調べることができる．このことは有望にも，ユーザに命令が要求される機能を果たしていることを保証するのに十分である．より複雑なデバッガは，実行される命令の正確な数の指定や各命令の実行に続いて表示されるレジスタ，あるいはメモリの指定ができる，シングル・ステップ・ルーチンの高度な形式を有している．
- ブレークポイントの機能：ブレークポイントの機能により，ユーザの指定した位置のオブジェクト・コードに，特殊なコード，8086の場合はソフトウエア・インタラプト命令を置くことによって，実行を制御することができる．特殊なコードが実行されると，その結果，制御がデバッガに移り，ユーザのオブジェクト・コードの実行が停止させられる．この時点で，デバッガは，特殊なコードに変更されていた位置に元のオブジェクト・コードを戻して，ＣＰＵの状態が調べられるようにする．

デバッガは一般に，メモリの任意の部分とＣＰＵの内部レジスタの内容を表示することができ，したがって状態を完全に調べられる．

より複雑なデバッガでは，次のことが可能である．

- メモリやレジスタの内容の変更
- オブジェクト・コードの実行のトレース
- オブジェクト・コードのアセンブルやディスアセンブル
- マス記憶からの読み込みや書き込み
- 簡単な算術演算機能の実行
- より高度なブレークポイントの実行
- シンボル・テーブルの操作

代表的なデバッガは，次のようなメモリやレジスタの内容変更に対して，いくつかの機能を備えている．

- メモリを調べて必要ならば変更する．

● 一連のメモリを定数で満たす.

● メモリ・ブロックの内容を他のメモリ・ブロックに移動する.

メモリを定数で満たすサンプルのデバッガ・コマンドが F $addr_1, addr_2$, 定数 である場合を考える. このコマンドは, $addr_1$ から $addr_2$ (このアドレスも含む) までのメモリのすべてを定数で満たす. たとえば, デバッガ・コマンド

```
F100,17F,20
```

が入力されると, デバッガは 100_{16} から $17F_{16}$ のすべての位置に定数 20_{16} を書き込む.

オブジェクト・コード実行のトレースを可能とするデバッガは, 一般にこの機能をシングル・ステップの拡張として行なう. ユーザは通常, 実行されるべきステップ数と表示されるべき情報の種類を指定して, これによってプログラムの実行を見る (トレースする) ことができる.

16進ではなくて, アセンブリ言語の命令としてメモリ内容の表示を指定できるような, メモリ表示のより高度な形式を備えているデバッガもある. たとえば

```
D   400,405
```

では

```
400   E4 10 24 40 74 FA
```

となるが, この代わりに

```
L   400
```

のコマンドは, メモリ 400_{16} — 405_{16} を次のように表示する.

```
400   IN AL.,10
402   AND AL,40
404   JZ 400
```

さらに, メモリ内容の変更に用いられる基本的なアセンブラの機能を備えているデバッガがある. 次のオブジェクト・コードを代入する代わりに,

```
S404   75,
405    FA
```

ソース・コードの代入ができる.

```
A404   JNZ
       400
```

デバッガはしばしば, マス記憶に対してデータの読み込みと書き込みを行なう基本的能力を備えている. 代表的能力には次のものがある.

● ユーザ指定のメモリ領域にマス記憶からオブジェクト・コード・ファイルを読み出す.

● ユーザ指定のメモリ領域からマス記憶にオブジェクト・コード・ファイルを書き込む.

● 基本的なペーパ・テープ処理の機能.

多くのデバッガは, 16進の算術演算機能を備えている. ユーザは通常, 2つの16進数値を入力して, デバッガに2つの数値の和と差を計算して表示させることができる.

より高度なブレークポイントを持つデバッガは, 一般に次のものを備えている.

● 各ブレークポイントのパス・カウント: ブレークポイント・アドレスから命令がフェ

ッチされる度に，パス・カウントは減少させられる．パス・カウントが0になれば，ユーザのプログラムは一時中止させられて，制御はデバッガに戻る．そしてユーザはＣＰＵの状態を見ることができる．この特徴は，プログラムのループをデバッグするときに特に有用となる．たとえば，53回目のループに何か問題が生じていると考えられるならば，53のパス・カウントでループ中にブレークポイントを設定して，妨げとなる繰り返しを都合のよいときに調べることができる．もしパス・カウントの機能が利用できなければ，53回の繰り返しの間にユーザのプログラム実行を停止させることは容易ではない．

- メモリがデータのためにアクセスされたときにユーザのオブジェクト・コードの実行を一時中止させる，ハードウエア・ブレークポイントの機能．これは，メモリが予測できないように壊されている場合に非常に有用となる．メモリ・アクセスのブレークにより，一般に問題の原因の識別が可能となる．

高度なデバッガは，アセンブラによって作られたシンボル・テーブルを用いて動作する．このデバッガでは，名前でメモリを参照することができる．この特徴は，リロケータブル・オブジェクト・コードで動作するときに非常に役立つ．マップとリストを用いて，特定の変数のアドレスを計算する代わりに，変数は名前で直接に参照することができる．

第6章
8086アセンブリ言語のプログラミング例

この章では，8086アセンブリ言語のプログラミングの2つの例，ソート・プログラムとI/Oドライバについて解説する．これら2つの例の仕様とプログラム設計の労苦は，既に2章に示されている．

6.1 ソート・プログラム

ソート・プログラムは，次の分離した3つのモジュールに分かれる．

- テープの読み込み
- レコードのソート
- テープの書き込み

“テープの読み込み”は次の1つのサブルーチンをコールしている．

- テープからのレコードの読み込み

“レコードのソート”は次の4つのサブルーチンをコールしている．

- サブソートを一時記憶領域へ移動
- キーの比較
- ポインタの計算
- レコードの移動

“テープの書き込み”は次の1つのサブルーチンをコールしている．

- テープへのレコードの書き込み

ソース・コードを見ると，サブルーチンの呼び出しのすべてが必要ではないことが明らかとなる．たとえば，“テープからのレコードの読み込み”ルーチンは，ソース・コードのただ1つのステートメントによってだけ呼び出されている．しかし，“テープからのレコードの読み込み”ルーチン全体が，それが呼び出されている位置のソース・コードに含まれている場合よりも，ソース・コードのモジュールはすっきりしている．

ソート・プログラムはアブノーマルの終了を2つ有している．この2つの終了は共に，テープ・コントローラの読み込み／書き込み・ルーチン中で発生する．このプログラムのためには，オペレーティング・システムの存在と，もし異常が検出されれば，オペレーティング・システムが適当な装置にエラー・メッセージを出力することが仮定されている．

```
;EQUATES FOR SORT ROUTINE

TAPE$COMMAND$PORT           EQU     20H             ;ARBITRARY #'S. TYPICALLY
TAPE$STATUS$PORT            EQU     20H             ;THESE #'S WOULD BE LISTED
TAPE$DATA$PORT              EQU     22H             ;IN THE SPECIFICATION.

READ$TAPE$COMMAND           EQU     01H             ;FROM SPECIFICATION
WRITE$TAPE$COMMAND          EQU     02H

OPERATION$COMPLETE$FLAG     EQU     04H
TAPE$ERROR$STATUS           EQU     080H

TAPE$ERROR$FLAG             EQU     044H            ;USED BY SYSTEM

;EXTERNAL REFERENCES

EXTRN SYSTEM:               FAR
EXTRN SYSTEM$ERROR:         FAR
```

```
DATA SEGMENT
;RAM LOCATIONS FOR SORT PROGRAM

RECORD$TEMP                 DB     2
KEY$TEMP                    DB     10

INDEX                       DW     1
INTERVAL                    DW     1
SUBSORT$COUNTER             DW     1
RECORD$COUNT                DW     1

TAPE$BUFFER                 DB     140 DUP(?)
SORT$AREA                   DB     4000 DUP (12 DUP(0))
DATA ENDS
```

```
CODE SEGMENT
ASSUME CS: CODE, DS: DATA, ES: DATA
MAIN:
                            MOV     AX,DATA                 ;LOAD SEGMENT REGISTERS
                            MOV     DS,AX
                            MOV     ES,AX
                            MOV     RECORD$COUNT,0          ;SET # OF RECORDS TO 0
                            MOV     DI,OFFSET SORT$AREA     ;POINT DI AT SORT AREA
```

```
; READ THE TAPE MODULE OPERATES BY
; 1.  READING A TAPE RECORD
; 2.  CHECKING FOR DONE
; 3.  MOVING 6 WORDS FROM THE TAPE BUFFER TO THE SORT AREA
; 4.  UPDATING THE # OF RECORDS
;
;
READ$THE$TAPE:              CALL    READ$TAPE$BUFFER        ;READ 128 BYTES
                            MOV     SI,OFFSET TAPE$BUFFER   ;TEST FOR EOF RECORD
                            CMP     [SI],OFFFFH             ;GO SORT IF EOF
                            JZ      SORT                    ;NOT EOF, MOVE RECORD #
                            MOV     CX,12                   ;AND KEY
                            REP     MOVS TAPE$BUFFER,       ;INCREMENT # OF RECORDS
                                    SORT$AREA
                            INC     RECORD$COUNT            ;GET ANOTHER BYTE
                            JMP     READ$THE$TAPE
```

ステートメント

```
MOV    SI,OFFSET  TAPE$BUFFER     ;TEST  FOR  EOF  RECORD
```

のOFFSETオペレータは，イミディエイト・データとしてSIレジスタにTAPE$BUFFERのアドレスをロードするオブジェクト・コードの生成に用いられる．

ステートメント

```
MOV    SI,TAPE$BUFFER
```

は，TAPE$BUFFERの内容をSIレジスタにロードするオブジェクト・コードを生成することに注意．OFFSETオペレータは標準のインテル8086アセンブラの特徴であり，8086マイクロプロセッサの特徴ではない．

奇数バイトの移動よりも，偶数バイトの移動が容易であることに注意．奇数バイトを移動するためには，2つの方法が考えられる．その1つを次に示す．

```
MOV    CX, ODD$NUMBER        ;LOAD  #  OF  BYTES
REP    MOVSB                 ;TO  MOVE
```

この方法では，偶数バイトの移動に用いられているのと同数バイトのオブジェクト・コードを用いているが，実行には2倍の時間を要する．もう1つの方法を次に示す．

```
MOV    CX, ODD$NUMBER        ;LOAD  #  OF  WORDS TO  MOVE
SHR    CX, 1
REP    MOVSW
MOVSB                        ;MOVE  LAST  BYTE
```

この方法は，さらに1バイトのオブジェクト・コードを必要とするが，偶数バイト移動ルーチンと同じ時間で実行される．

```
;  THE SORT MODULE IS A STRAIGHTFORWARD RENDITION OF THE ALGORITHM
;  PRESENTED IN CHAPTER 3
SORT:                     MOV   AX,RECORD$COUNT      ;INITIALIZE INTERVAL TO
                          MOV   INTERVAL,AX          ;RECORD COUNT
NEW$INTERVAL:             SHR   INTERVAL,1           ;DIVIDE INTERVAL BY 2
                          JZ    WRITE$TO$TAPE
                          MOV   AX,RECORD$COUNT      ;SUBSORT CTR=RECORD$
                                                      COUNT- INTERVAL
                          SUB   AX,INTERVAL
                          MOV   SUBSORT$COUNTER,AX
NEXT$SUBSORT$COUNTER:     INC   SUBSORT$COUNTER
                          MOV   AX,SUBSORT$COUNTER
                          CMP   AX,RECORD$COUNT      ;TEST FOR NEW INTERVAL
                          JG    NEW$INTERVAL
                          CALL  MOVE$SUBSORT$ TO$TEMP ;SAVE CURRENT RECORD
                          MOV   AX,SUBSORT$COUNTER   ;INDEX=SUBSORT CTR-INTERVAL
                          SUB   AX,INTERVAL
TEST$KEYS:                MOV   INDEX,AX
                          CALL  COMPARE$KEYS
                          JGE   FOUND$THIS$RECORDS $SPOT
                          MOV   AX,INDEX
                          CALL  COMPUTE$POINTER
                          MOV   SI,AX
                          CALL  MOVE$RECORD
```

```
                                  MOV   AX,INTERVAL              ;INDEX-INTERVAL=INDEX
                                  SUB   INDEX,AX
                                  JGE   TEST$KEYS
FOUND$THIS$RECORDS$SPOT:          MOV   SI,OFFSET RECORD$TEMP
                                  CALL  MOVE$RECORD
                                  JMP   NEXT$SUBSORT$COUNTER
```

命令

```
SHR    INTERVAL,1
```

は,

```
MOV    AX,INTERVAL
SHR    AX,1
MOV    INTERVAL,AX
```

よりも，メモリ使用と時間消費の両方からより効率的であることに注意．すべての場合，情報をレジスタに移動し，それを操作して結果をメモリに戻すよりも，直接にメモリで処理する方がより能率がよい．

```
;   WRITE TAPE OPERATES BY
;   1.  INITIALIZING PTRS TO THE TAPE BUFFER AND SORT AREA
;   2.  MOVING 12 BYTES AT A TIME UNTIL EITHER
;
;
;      128 BYTES HAVE BEEN MOVED
;      END OF FILE IS REACHED
;
;   3.  IF 128 BYTES, WRITE A TAPE RECORD
;   4.  IF END OF FILE, APPEND AN EOF RECORD, THEN WRITE
;   THE LAST TAPE RECORD
;
WRITE$TO$TAPE:          MOV   SI,OFFSET SORT$AREA
NEXT$TAPE$BUFFER:       MOV   DI,OFFSET TAPE$BUFFER
MOVE$NEXT$RECORD:       MOV   CX,12                        ;GET READY TO MOVE 12 BYTES
                        REP   MOVS  TAPE$BUFFER,SORT$AREA
                        CMP   DI,OFFSET TAPE$BUFFER + 128
                        JL    UPDATE$RECORD$COUNT          ;TEST FOR MOVED FULL BUFFER
                        PUSH  SI                           ;SAVE POINTERS
                        PUSH  DI
                        CALL  WRITE$TAPE$BUFFER            ;WRITE 128 BYTES TO TAPE
                        POP   DI                           ;RESTORE POINTERS
                        POP   SI
                        MOV   AX,OFFSET TAPE$BUFFER + 128  ;ANY EXTRAS IN END OF TAPE
                        SUB   DI,AX                        ;BUFFER
;   NOTE: TO FILL 128 BYTES REQUIRES MOVING 11 RECORDS OR 11 X 12 =
;   132 BYTES INTO TAPE BUFFER
                        MOV   CX,DI                        ;CX GETS COUNT
                        MOV   DI,OFFSET TAPE$BUFFER
                        JZ    UPDATE$RECORD$COUNT          ;JUMP IF NO EXTRAS
                        PUSH  SI                           ;SAVE POINTER INTO SORT AREA
                        MOV   SI,AX
                        REP   MOVS TAPE$BUFFER,TAPE$BUFFER ;MOVE EXTRAS DOWN TO START
                              POP     SI                   ;OF TAPE BUFFER
UPDATE$RECORD$COUNT:    DEC   RECORD$COUNT
                        JNZ   MOVE$NEXT$RECORD
```

```
                CMP   DI,OFFSET TAPE$BUFFER          ;TEST IF ONE MORE RECORD
                JZ    WRITE$EOF                      ;MUST BE WRITTEN BEFORE EOF
                MOV   CX, OFFSET TAPE$BUFFER + 128
                SUB   CX,DI                          ;ZERO OUT THE REST OF
                XOR   AL,AL                          ;THE TAPE BUFFER
                REP   STOS TAPE$BUFFER
                CALL  WRITE$TAPE$BUFFER              ;WRITE LAST TAPE RECORD.
WRITE$EOF:      MOV   TAPE$BUFFER,OFFFFH             ;MOVE IN END OF FILE RECORD
                CALL  WRITE$TAPE$BUFFER              ;WRITE EOF RECORD, A
                JMP   SYSTEM                         ;RECORD WITH FFFF IN
                                                     ;THE FIRST TWO BYTES
                                                     ;END OF PROGRAM
                                                     ;RETURN TO SYSTEM
```

```
;PROCEDURES CALLED BY MAIN PROGRAM

COMPUTE$POINTER       PROC   NEAR
   AX HAS INDEX
   RETURN ADDR IS IN AX
   DX IS NOW 0
                      MOV    CX,12
                      MUL    CX
                      ADD    AX,OFFSET SORT$AREA          ;ADD ADDRESS, NOT DATA

                      RET
COMPUTE$POINTER       ENDP
```

このモジュールは，

```
MOV     CX,12
MUL     CX
```

を

```
SHL     AX,1
SHL     AX,1
MOV     CX, AX
SHL     AX,1
ADD     AX,CX
```

で置き換えることによって処理速度を上げることができる．

MOV/MULの命令は，実行に126サイクルを要する．2番目の命令は実行に11サイクルを要し，しかもDXレジスタを壊さない．ただし，MOV/MULの命令ではプログラム・メモリの5バイトだけでよいのに対して，2番目の命令は10バイトを要する．

```
;  WRITE TAPE BUFFER OPERATES BY
;  1.  POINTING AT THE TAPE BUFFER
;  2.  INITIALIZING THE TAPE CONTROLLER FOR A WRITE
;  3.  CHECKING FOR STATUS ERRORS
;  4.  CHECKING FOR OPERATION DONE
;  5.  WRITING TO THE TAPE DATA PORT

WRITE$TAPE$BUFFER    PROC   NEAR
                     MOV    SI,OFFSET TAPE$BUFFER        ;GET ADDRESS OF TAPE BUFFER
                     MOV    AL,WRITE$TAPE$COMMAND        ;START TAPE WRITE
                     OUT    TAPE$COMMAND$PORT,AL

GET$TAPE$STATUS:     IN     AL,TAPE$STATUS$PORT          ;CHECK FOR ERRORS
                     TEST   AL,TAPE$ERROR$STATUS
```

```
                        JNZ     OUTPUT$TAPE$ERROR
                        TEST    AL,OPERATION$COMPLETE$FLAG ;TEST FOR DONE
                        JNZ     WRITE$COMPLETE
                        LODSB                              ;GET A BYTE
                        OUT     TAPE$DATA$PORT,AL          ;SHIP IT OUT
                        JMP     GET$TAPE$STATUS
OUTPUT$TAPE$ERROR:      MOV     AH,TAPE$ERROR$FLAG
                        JMP     SYSTEM$ERROR
WRITE$COMPLETE:         RET
WRITE$TAPE$BUFFER       ENDP
```

ＴＥＳＴの操作

```
                        TEST    AL,TAPE$ERROR$STATUS
```

は，ＡＬレジスタのステータス・バイトを次の操作のために保存するために，ＡＮＤ操作の代わりに用いられる．

ＡＬレジスタの内容は，調べる操作が終了した後では意味を持たないので，

```
                        TEST    AL,OPERATION$COMPLETE$FLAG
```

の操作は

```
                        AND     AL,OPERATION$COMPLETE$FLAG
```

で置き換えることができる．

```
;   READ TAPE BUFFER OPERATES BY
;   1. INITIALIZING TAPE CONTROLLER TO READ
;   2. CHECKING FOR TAPE ERRORS
;   3. CHECKING FOR COMPLETION
;   4. READING DATA FROM TAPE DATA PORT
;
;   THIS ROUTINE USES SI AND AL
;
;   IF AN ERROR OCCURS, THIS ROUTINE BRANCHES TO THE SYSTEM
;
;
;
READ$TAPE$BUFFER        PROC    NEAR
                        MOV     SI,OFFSET TAPE$BUFFER        ;POINT AT TAPE BUFFER
                        MOV     AL,READ$TAPE$COMMAND         ;TELL TAPE TO READ
                        OUT     TAPE$COMMAND$PORT,AL
GET$STATUS:             IN      AL,TAPE$STATUS$PORT
                        TEST    AL,TAPE$ERROR$STATUS         ;CHECK FOR TAPE ERRORS
                        JNZ     TAPE$ERROR
                        TEST    AL,OPERATION$COMPLETE$FLAG   ;CHECK FOR DONE
                        JNZ     READ$COMPLETE
                        IN      AL,TAPE$DATA$PORT            ;GET DATA
                        MOV     [SI],AL                      ;SAVE DATA
                        INC     SI
                        JMP     GET$STATUS
TAPE$ERROR:             MOV     AH,TAPE$ERROR$FLAG           ;CALL SYSTEM
                        JMP     SYSTEM$ERROR                 ;ERROR PROCESSOR
READ$COMPLETE:          RET
READ$TAPE$BUFFER        ENDP
```

このルーチンとWRITE$TAPE$BUFFERルーチンとの類似性に注意．読み込みと書き込みのルーチンでは，2つの異なる点がある．

```
        WRITE$TAPE$BUFFER     MOV     AL,WRITE$TAPE$COMMAND
```

は

```
        READ$TAPE$BUFFER      MOV     AL,READ$TAPE$COMMAND
```

で置き換えられ，

```
                              LODSB
                              OUT      AL,TAPE$DATA$PORT
```

は

```
                              IN      AL,TAPE$DATA$PORT
                              MOV     [SI],AL
                              INC     SI
```

で置き換えられている．

読者は練習として，テープ・バッファの読み込みと書き込みのルーチンが共通コードを共有するように，この2つのルーチンを1つにしてみよ．テープ・バッファのポインタとしてDIレジスタを用いればより効果的となることに注意．たとえば，

```
                              MOV     [SI],AL
                              INC     SI
```

は

```
                              STOSB
```

で置き換えられる．ただし，DIレジスタはメインのREADTHETAPEモジュールで用いられている．

```
; COMPARE KEYS OPERATES BY COMPARING KEY (INDEX) WITH KEYTEMP
; 1. CALCULATE INDEX
; 2. COMPARE KEYS UNTIL
;      • DIFFERENCE IS FOUND
;      • 10 BYTES HAVE BEEN COMPARED
COMPARE$KEYS        PROC    NEAR
                    MOV     AX,INDEX                ;GET INDEX
                    CALL    COMPUTE$POINTER
                    INC     AX                      ;POINT PAST RECORD #
                    INC     AX
                    MOV     DI,AX                   ;MOVE TO DI FOR COMPARE
                    MOV     SI,OFFSET KEY$TEMP
                    MOV     CX,0010                 ;10 BYTES TO COMPARE
                    CMPS    KEY$TEMP, SORT$AREA
                                                    ;COMPARE 5 WORDS
                    RET
COMPARE$KEYS        ENDP
```

命令

```
                    MOV      SI,OFFSET     KEY$TEMP
```

は，KEY$TEMPの値ではなく，KEY$TEMPのアドレスをSIレジスタにロードする．

```
;  MOVE RECORD OPERATES BY MOVING WHATEVER SI POINTS AT TO THE LOCATIONS
;  POINTED TO BY INDEX INTERVAL
;  1.  CALCULATE PTR. FOR INDEX + INTERVAL
;  2.  MOVE 12 BYTES
MOVE$RECORD          PROC    NEAR
                     MOV     AX,INDEX
                     ADD     AX,INTERVAL                   ;CALC INDEX + INCREMENT
                     CALL    COMPUTE$POINTER
                     MOV     DI,AX
                     REP     MOVS SORT$AREA,SORT$AREA      ;COMPUTER POINTER
                                                            RETURNS CX=12
                     RET
MOVE$RECORD          ENDP
```

メモリ領域に対して時間の節約が必要ならば,

```
MOV     DI,AX
REP     MOVSB
RET
```

の命令は

```
MOV     DI,AX
SHR     CX,1
REP     MOVSW
RET
```

で置き換えられる.

MOVSBをMOVSWで置き換えることにより,6×17=102クロックが節約される.シフト命令に必要な2サイクルを減じると,正味100サイクルの節約が実現される.

オペランドの指定が,そのオペランドのタイプからバイトあるいはワードの操作を決定するアセンブラがあることに注意.この例では,

```
MOVS     SORT$AREA, SORT$AREA
```

は,SORT$AREAがバイト構成のバッファなので,バイト操作となる.MOVSのMOVSBとMOVSWのタイプは,オペランドの指定とタイプの判断を無視して,アセンブラに何を行なうかを知らせる.簡潔で理解しやすいコードのために,前者の方法はしばしば好ましいが,後者は上の例のように効力を無効にする.

```
;  MOVE SUBSORT TO TEMP OPERATES BY:
;  1.  CALCULATING SUBSORT PTR.
;  2.  LOADING POINTER TO RECORD TEMP.
;  3.  MOVING BYTES
MOVE$SUBSORT$TO$TEMP    PROC    NEAR
                        MOV     AX,SUBSORT$COUNTER
                        CALL    COMPUTE$POINTER
                        MOV     SI,AX
                        MOV     DI,OFFSET RECORD$TEMP
                        REP     MOVSB                  ;COMPUTE POINTER RETURNS
                                                       ;CX = 12
                        RET
MOVE$SUBSORT$TO$TEMP    ENDP
```

前のモジュールと同様に，このコードの実行時間は，

```
REP     MOVSB
```

を

```
SHR     CX,1
REP     MOVSW
```

で置き換えることによって減少できる．

6.2 I/Oドライバ

I/Oドライバのプログラムのコーディングには次の2つの基本的問題がある．

- これを利用しようとする外部ルーチンによってどのように呼び出されるか．
- このルーチンにパラメータはどのように受け渡されるか．

このルーチンは次の3つの基本的な方法で呼び出される．

- オペレーティング・システムをコールする．これにより，要求は適当なモジュールに受け渡される．
- ドライバ全体のコマンド・ハンドラをコールする．これにより，パラメータが選択されて制御が適当なモジュールに渡される．
- 呼び出し元ルーチンのコード内に存在するアドレスを用いて，直接にルーチンをコールする．

この例では，ルーチンは直接にコールされることを仮定している．呼び出し元プログラムは各ルーチンのエントリ・アドレスを知っている．これは，最初の2つの方法の利用を除外しない．もしこのどちらかが都合がよければ，各モジュールを示す簡単な発送テーブルによって，オペレーティング・システムまたはコマンド・ハンドラはコマンドを適当なモジュールに分配できる．

2章で述べたように，パラメータの受け渡しには3つの方法がある．

- レジスタ
- タスク・ブロック
- スタック

この例では，複数キャラクタの入力と出力のルーチンの場合を除いて，パラメータはレジスタで受け渡される．この機能のために，タスク・ブロックを定義する．複数キャラクタの入力では，次のタスク・ブロックを用いる．

#0:		リードするキャラクタの最大数
#1:		実際にリードしたバイト数
#2:		バイト2からnまでは，複数キャラクタの入力ルーチンによってリードされた情報を含む．
⋮		
#n:		

複数キャラクタの出力では，次のタスク・ブロックを用いる．

#0:		出力すべきバイト数
#1:		バイト1からnは，データ・ポートのチャンネルに送られる
⋮		
#n:		

```
CONTROL$PORT                        EQU     12H
STATUS$PORT                         EQU     12H
DATA$PORT                           EQU     10H

;  IF BIT 0 OF THE AH REGISTER IS 1, THE USER HAS LOADED SI WITH A
;  POINTER TO THE STRING TO BE SENT TO THE CONTROL PORT  IF BIT 0 IS A 0, A
;  STANDARD INITIALIZATION STRING WILL BE SENT
;
USER$INITIALIZATION$BIT             EQU     01H
TIMEOUT$VALUE                       EQU     0F000H

;  BITS 3, 4, AND 5 OF THE SIO STATUS BYTE ARE ERROR BITS

SIO$ERRORS                          EQU     38H
;  BIT 1 INDICATES WHETHER OR NOT THE RECEIVER IS READY
;  BIT 0 INDICATES WHETHER OR NOT THE TRANSMITTER IS READY

SIO$RECEIVER$READY                  EQU     02H
SIO$TRANNY$READY                    EQU     01H
TIMEOUT$ERROR$FLAG                  EQU     0FFH

;  CARRIAGE RETURN IS TERMINATION CHARACTER FOR READ

CARRIAGE$RETURN                     EQU     0DH

;  '$' IS TERMINATION CHARACTER FOR WRITE

TERMINATION$CHARACTER               EQU     24H
EXTRN SYSTEM$ERROR:                 FAR
```

```
CODE SEGMENT
  ASSUME CS: CODE

;  THE INITIALIZATION OPERATES BY:
;  1.  TESTING FOR USER SPECIFIED OR
;       SYSTEM INITIALIZATION STRING
;  2.  SENDING THE STRING TO THE CONTROL PORT,
;       TERMINATING WHEN A 0 IS DETECTED
;
;  THIS ROUTINE USES AX AND SI

INITIALIZATION          PROC    NEAR
                        AND     AH,USER$INITIALIZATION$BIT          ;TEST FOR USER INIT
                        JNZ     SI$LOADED$BY$USER
                        MOV     SI,OFFSET PORT$INITIALIZATION$STRING ;LOAD STANDARD STRING
SI$LOADED$BY$USER:      LODSB
                        OR      AL,AL                               ;SET FLAGS TO TEST FOR 0
                        JZ      DO$A$RETURN                         ;EXIT IF 0
                        OUT     CONTROL$PORT,AL
                        JMP     SI$LOADED$BY$USER

PORT$INITIALIZATION$STRING      DB      0CEH,40H,0CEH,37H,00H
DO$A$RETURN:                    RET
INITIALIZATION                  ENDP
```

このルーチンが呼ばれたときの8251は既知の状態にはない事実を考慮すると，4バイトの初期設定ストリングが必要である．たとえば，

CE $_{16}$ モード
37 $_{16}$ コマンド

の2バイトが8251に送られると，コマンド・コントロール入力を待っている可能性もあるので，8251は正しく初期設定されないときがある．もし

40 $_{16}$ コマンド（リセット）
CF $_{16}$ モード
37 $_{16}$ コマンド

の3バイトが送られると，モード・コントロール入力を待っていた場合には，8251は正しく初期設定されない．しかし，4バイトを用いれば，以前の状態にかかわらず，8251を正しく初期設定できる．

```
;  SINGLE CHARACTER INPUT OPERATES BY:
;  1.  LOADING TIMEOUT VALUE
;  2.  READING THE STATUS PORT AND TESTING FOR SIO ERRORS
;  3.  CHECKING FOR TIMEOUT ERRORS
;  4.  READING THE DATA
;
;  THIS ROUTINE USES AX AND CX
;
;  IF ZFLAG IS 1 ON RETURN - ERROR CONDITION
;  IF ZFLAG IS 0 ON RETURN - NORMAL OPERATION
;  ERROR CONDITIONS RETURNED IN AH
SINGLE$CHARACTER$INPUT     PROC    NEAR
                           MOV     CX,TIMEOUT$VALUE
TEST$STATUS:               IN      AL,STATUS$PORT              ;READ STATUS
                           TEST    AL,SIO$ERRORS               ;CHECK FOR ERRORS
                           JNZ     INPUT$ERROR$RETURN
                           DEC     CX                          ;CHECK FOR TIMEOUT
                           JZ      INPUT$TIMEOUT$ERROR$RETURN
                           AND     AL,SIO$RECEIVER$READY       ;RECEIVER READY?
                           JZ      TEST$STATUS
                           IN      AL,DATA$PORT                ;GET VALUE
                           RET
INPUT$ERROR$RETURN:        MOV     AH,AL                       ;SAVE STATUS
                           XOR     AL,AL                       ;SET ZERO FLAG
                           RET
INPUT$TIMEOUT$ERROR$       MOV     AH,TIMEOUT$ERROR$FLAG       ;FF IS TIMEOUT ERROR
 RETURN:                   RET
SINGLE$CHARACTER$INPUT     ENDP
```

```
DEC     CX
JZ      INPUT$TIMEOUT$ERROR$RETURN
AND     AL,SIO$RECEIVER$READY
```

の命令は

```
                        LOOP    NO$TIMEOUT
                        MOV     AH,0FFH                 ;TIMEOUT  ERROR
                        RET
    NO$TIMEOUT:         AND     AL,SIO$RECEIVER$READY
```

で置き換えられる．これにより，短くて速いオブジェクト・コードが得られる．ただし，ソース・コードの明瞭さを犠牲にしている．

```
;  SINGLE CHARACTER OUTPUT OPERATES BY:
;  1.  LOADING TIMEOUT VALUE
;  2.  READING STATUS PORT
;  3.  CHECKING FOR TIMEOUT ERROR
;  4.  SENDING DATA TO OUTPUT PORT IF TRANSMITTER IS READY
;
;  IF ZFLAG IS 1 ON RETURN - ERROR
;  IF ZFLAG IS 0 ON RETURN - NORMAL
;
;  THIS ROUTINE USES AX, CX, AND DH
SINGLE$CHARACTER$OUTPUT   PROC   NEAR
                          MOV    CX,TIMEOUT$VALUE
                          MOV    DH,AL
TRANNY$READY:             IN     AL,STATUS$PORT
                          TEST   AL,SIO$ERRORS
                          JNZ    OUTPUT$ERROR$RETURN
                          DEC    CX                     ;TEST FOR TIMEOUT
                          JZ     OUTPUT$TIMEOUT$ERROR$RETURN
                          AND    AL,SIO$TRANNY$READY    ;CHECK FOR TRANSMITTER READY
                          JZ     TRANNY$READY
                          MOV    AL,DH                  GET DATA FROM DH
                          OUT    DATA$PORT,AL
                          RET
OUTPUT$TIMEOUT$ERROR$     MOV    AH,TIMEOUT$ERROR$FLAG
 RETURN:                  RET
OUTPUT$ERROR$RETURN:      MOV    AH,AL
                          XOR    AL,AL
                          RET
SINGLE$CHARACTER$OUTPUT   ENDP
```

前のモジュールと同様に，タイムアウトのエラー・リターンはメインのコードに含むことができる．

```
                        DEC     CX
                        JZ      OUTPUT$TIMEOUT$ERROR$RETURN
                        AND     AL,SIO$TRANNY$READY
```

を

```
                        LOOP    NO$TIMEOUT
                        MOV     AH,0FFH
                        RET
    NO$TIMEOUT:         AND     AL,SIO$TRANNY$READY
```

で置き換えて，ソース・コードの最後の3行を削除することによって行なえる．

```
CHECK$CHANNEL$STATUS          PROC    NEAR
                              IN      AL,STATUS$PORT              ;READ
                              RET
CHECK$CHANNEL$STATUS          ENDP
```

```
SEND$CONTROL$INFORMATION      PROC    NEAR
                              OUT     CONTROL$PORT,AL             ;WRITE
                              RET
SEND$CONTROL$INFORMATION      ENDP
```

```
;  MULTIPLE CHARACTER INPUT OPERATES BY:
;  1.  GETTING # OF BYTES TO READ
;  2.  CALLING SINGLE CHARACTER INPUT UNTIL
;      ・ ERROR FROM SINGLE CHAR
;      ・ THE MAXIMUM # OF CHARACTERS HAVE BEEN ENTERED
;      ・ A TERMINATION CHARACTER (CARRIAGE RETURN) IS ENTERED
;  THIS ROUTINE IS CALLED WITH SI POINTING AT THE TASK BLOCK
;
;THIS ROUTINE USES SI, DI, AX, CX

MULTIPLE$CHARACTER$INPUT      PROC    NEAR
                              LODSB
                              OR      AL,AL                       ;LOAD MAX # OF BYTES TO READ
                              JZ      ZERO$COUNT$THEN$RETURN
                              MOV     DL,AL                       ;SAVE MAX # IN DL
                              MOV     DI,SI
                              INC     DI                          ;POINT AT BUFFER
GET$A$CHARACTER:              CALL    SINGLE$CHARACTER$INPUT      ;GET CHARACTER
                              JZ      INPUT$ERROR
                              STOSB
                              INC     BYTE PRT [SI]               ;INCREMENT # READ
                              CMP     DL,[SI]                     ;TEST FOR READ MAXIMUM #
                              JZ      ZERO$COUNT$THEN$RETURN

                              CMP     AL,CARRIAGE$RETURN
                              JNZ     GET$A$CHARACTER
ZERO$COUNT$THEN$RETURN:       RET
INPUT$ERROR:                  JMP     SYSTEM$ERROR
MULTIPLE$CHARACTER$INPUT      ENDP
```

```
                              OR      AL,AL
                              JZ      ZERO$COUNT$THEN$RETURN
```

の命令は，読み込むべきバイト数が0かを調べる．

1つの命令でデータのセーブとポインタの増加を行なうストリング・プリミティブSTOSBのために，DIは入力バッファのポインタとして用いられる．ここでは，ディレクション・フラグは正しく設定されていると仮定している．

```
;  MULTIPLE CHARACTER OUTPUT OPERATES BY:
;  1.  GETTING THE NUMBER OF CHARACTERS TO WRITE
;  2.  CALLING SINGLE CHARACTER OUTPUT UNTIL
;      ・ ERROR FROM SINGLE CHARACTER OUTPUT
;      ・ THE MAXIMUM # OF CHARACTERS HAVE BEEN WRITTEN
;
;  THIS ROUTINE IS CALLED WITH SI POINTING AT THE TASK BLOCK
;
;  THIS ROUTINE USES AX, SI, DX, AND CX
MULTIPLE$CHARACTER$OUTPUT     PROC    NEAR
                              LODSB
                              OR      AL,AL
                              JZ      DO$RETURN
                              MOV     DL,AL                       ;SAVE # OF BYTES TO
                                                                   OUTPUT
```

```
OUTPUT$A$CHARACTER:          LODSB
                             CMP     AL,TERMINATION$CHARACTER
                             JZ      DO$RETURN                  ;TEST FOR TERMINATION
                                                                 CHARACTER
                             CALL    SINGLE$CHARACTER$OUTPUT
                             JZ      OUTPUT$ERROR
                             DEC     DL
                             JNZ     OUTPUT$A$CHARACTER
DO$RETURN:                   RET
OUTPUT$ERROR:                JMP     SYSTEM$ERROR
MULTIPLE$CHARACTER$OUTPUT    ENDP
```

第7章
8086マイクロプロセッサ

この章で初めて，8086と外部ロジックとのインターフェイスの方法について述べる．したがって，この章では，8086が発生または受け取る信号について検討し，8086システムの概念について概要を調べることにする．

この章と以後の章では，5 MHzの標準的な8086のタイミングを主に取り扱っている．A.C．パラメータに適当な代入を行なうことによって，8と10MHzのバージョンにも，示されている式は適用できる．

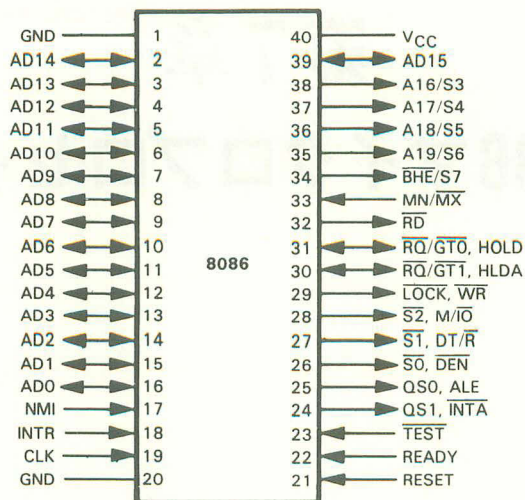
7.1　8086ＣＰＵのピンと信号

8086ＣＰＵのピンと信号を図7-1に示す．特に示されている信号を除いて，すべての入力と出力はＴＴＬレベルと互換性がある．

すべてのマイクロプロセッサは，次の種類の信号を生成または受け取る．

- ●アドレス・ライン
- ●データ・ライン
- ●コントロールとステータスのライン
- ●パワーとタイミングのライン

8086の40ピン・パッケージは，４つのタイプすべての信号を有する．いくつかのピンは，１つ以上のタイプの情報を運ぶ．たとえば，データとアドレスのラインは多重化されている．他のピンは，$MN/\overline{MX}$ピンのレベルで定義される機能を持つ．

以下の解説では，$MN/\overline{MX}$がハイあるいはローのとき，各ピンの機能について述べている．

ピンの名前	種　　類	型
AD0-AD15	データ/アドレス・バス	双方向，トライステート
A16/S3, A17/S4	アドレス/セグメント識別子	出力，トライステート
A18/S5	アドレス/インタラプト・イネーブル・ステータス	出力，トライステート
A19/S6	アドレス/ステータス	出力，トライステート
$\overline{\text{BHE}}$/S7	上位バイト/ステータス	出力，トライステート
$\overline{\text{RD}}$	リード・コントロール	出力，トライステート
READY	ウエート状態リクエスト	入力
$\overline{\text{TEST}}$	テストのウエート・コントロール	入力
INTR	インタラプト・リクエスト	入力
NMI	ノンマスカブル・インタラプト・リクエスト	入力
RESET	システム・リセット	入力
CLK	システム・クロック	入力
MN/$\overline{\text{MX}}$	マキシマム・システムでは＝GND	
$\overline{\text{S0}}$, $\overline{\text{S1}}$, $\overline{\text{S2}}$	マシーン・サイクル・ステータス	出力，トライステート
$\overline{\text{RQ}}/\overline{\text{GT0}}$, $\overline{\text{RQ}}/\overline{\text{GT1}}$	ローカル・バス・プライオリティ・コントロール	双方向
QS0, QS1	インストラクション・キュー・ステータス	出力
$\overline{\text{LOCK}}$	バス・ホールド・コントロール	出力，トライステート
MN/$\overline{\text{MX}}$	ミニマム・システムでは＝V_{CC}	
M/$\overline{\text{IO}}$	メモリまたはI/Oのアクセス	出力，トライステート
$\overline{\text{WR}}$	ライト・コントロール	出力，トライステート
ALE	アドレス・ラッチ・イネーブル	出力
DT/$\overline{\text{R}}$	データのトランスミット/レシーブ	出力，トライステート
$\overline{\text{DEN}}$	データ・イネーブル	出力，トライステート
$\overline{\text{INTA}}$	インタラプト・アクノリッジ	出力，トライステート
HOLD	ホールド・リクエスト	入力
HLDA	ホールド・アクノリッジ	出力
V_{CC}, GND	パワー，グランド	

（濃い網掛け）マキシマム・システム信号　（薄い網掛け）ミニマム・システム信号

図7-1　8086のピンと信号の割当て

7.1.1　アドレスとデータのライン

8086CPUは,100万（1メガ）バイトのメモリを直接にアドレス指定できる．これは，20ビットのアドレス情報が必要であることを意味する．

8086CPUは，上位バイトと下位バイトとして取り扱う16ビット単位でデータをアクセスする．

20ビットのアドレス・バスと16ビットのデータ・バスを40ピン・パッケージで可能とするために，データ・バスはアドレス・バスの下位16ビットと多重化されている．さらに4つのアドレス・ラインはステータス情報と多重化されている．アドレス／データとアドレス／ステータスの信号は次のとおりである．

AD0-AD15
A16/S3
A17/S4
A18/S5
A19/S6
$\overline{BHE}$/S7

これらのラインについて詳細に検討してみる．

AD0-AD15. この16本のラインは，アドレス・バスとデータ・バスの多重化されたラインである．バス・サイクルの最初のクロックの期間，このラインはアドレスの下位16ビットを含む．他のすべてのクロック・サイクルの間，このラインはデータ・バスとして用いられる．このラインは，8086がインタラプト・アクノリッジのサイクルあるいは“ホールド・アクノリッジ”のサイクルを実行しているときは，ハイ・インピーダンスの状態に置かれる．

A16/S3. 命令実行の最初のクロックの期間，このラインはアドレス16のラインとして動作する．I/O命令が実行されると,このラインは最初のクロックの期間，ローになる．他のすべてのクロックの期間，このラインはA17/S4と共にステータス情報を得るために用いられる．

A17/S4. 命令実行の最初のクロックの期間，このラインはアドレス17のラインとして動作する．I/O命令が実行されると，このラインは最初のクロックの期間，ローになる．他のサイルでは，このラインはA16/S3と共にステータス情報を得るために用いられる．

最初を除くすべてのクロックの期間，A16/S3とA17/S4は，次に示すように，どのセグメント・レジスタが8086のアドレスのセグメント部を生成しているかを示す情報を与える．

A17/S4	A16/S3	意　味
0	0	エキストラ・セグメント
0	1	スタック・セグメント
1	0	コード・セグメントまたはセグメントでない
1	1	データ・セグメント

この情報は，各セグメント・レジスタがそれぞれ独自の１メガバイトのメモリを指定するように，外部ロジックによって8086のメモリ領域を拡張するために利用できるが，このとき，異なるセグメントで計算されるメモリ・アドレスはオーバラップすることができなくなる．

A18／S5．命令実行の最初のクロックの期間，このラインはアドレス18のラインとして動作する．I/O命令が実行されると，このラインは最初のクロックの期間，ローになる．他のすべてのクロックの期間，このラインは8086のインタラプト・イネーブル・フラグの状態を示している．

A19／S6．命令実行の最初のクロックの期間，このラインはアドレス19のラインとして動作する．I/O命令が実行されると，このラインは最初のクロックの期間，ローになる．他のすべてのサイクルでは，8086は，システム・バスを制御していれば，このラインをローに保つ．"ホールド・アクノリッジ" の期間，他のバス・マスタがシステム・バスの制御を得られるように，8086はこのラインをフロート状態にする．

$\overline{\text{BHE}}$／S7．命令実行の最初のクロックの期間，このラインは$\overline{\text{BHE}}$として用いられる．データ・バスの上位８ビットでデータが転送される，読み出し，書き込み，そしてインタラプト・アクノリッジのシーケンスの間，$\overline{\text{BHE}}$はローに保たれる．この信号は，ＡＤ０のラインと共にメモリ・バンクを選択するために用いられる．8086のメモリ選択のより一般的な解説は次の節に示す．２番目以降のクロックの期間では，$\overline{\text{BHE}}$／Ｓ７は最初のクロックにおける出力レベルを維持する．

7.1.2　コントロールとステータスのライン

8086のコントロールとステータスのラインは２つに分類することができる．１つはMN／$\overline{\text{MX}}$ピンのレベルの影響を受けないものであり，もう１つはその機能がMN／$\overline{\text{MX}}$ピンの値に依存するものである．影響を受けないものには次のものがある．

$\overline{\text{RD}}$
READY
$\overline{\text{TEST}}$
INTR
NMI
RESET

$\overline{\text{RD}}$は，ＣＰＵがメモリあるいはI/O素子からデータを読み出すとき，ローが出力される．$\overline{\text{S2}}$-(M/$\overline{\text{IO}}$) ピンは，メモリまたはI/Oのどちらが要求されているかを指定する．

READYは，データ転送操作を行なう用意のできていることを示すために，選ばれたメモリまたはI/O素子によって用いられる．信号（$\overline{\text{RDY1}}$または$\overline{\text{RDY2}}$）は8284クロック・ジェネレータに入力され，READYの入力はクロックとの同期がとられる．適当なときにREADYにローが入力されると，READYがハイになるまで，8086は"ウエート"状態となる．**$\overline{\text{TEST}}$**は，8086のWAIT命令だけで用いられる入力である．WAIT命令が実行されると，8086は$\overline{\text{TEST}}$にローが入力されるまで休止する．

INTRは，割り込み要求の入力である．この信号は，各々の命令実行の最終クロックの期間に8086によってサンプルされる．インタラプト・イネーブル・ビットが1でINTRがハイならば，8086はインタラプト・アクノリッジ・シーケンスを実行して，制御を適当なインタラプト・サービス・ルーチンに移す．それ以外では，次の命令が実行される．INTRはレベル・トリガの入力である．

NMIは，ノンマスカブル・インタラプト要求の入力である．NMIはエッジ・トリガの入力である．NMIがローからハイになれば，8086は現在の命令の実行を終了して，制御をノンマスカブル・インタラプト・サービス・ルーチンに移す．ノンマスカブル・インタラプト・サービス・ルーチンのアドレスはメモリ位置00008_{16}に存在する．ソフトウエアではこのインタラプトを無効にできない．

RESETはシステム・リセット信号であり，パワーアップ時を除いてRESETが少なくとも50μs持続しなければならないときは，少なくとも4つのCLKクロックの期間，8284クロック・ジェネレータにハイが入力される必要がある．8284はRESETの同期をとり，8086に転送する．RESETがローに戻ると，次のことが起きる．

1．フラグ・レジスタが0000_{16}に設定される．これはインタラプトとシングル・ステップ・モードを無効にする作用をもつ．
2．DS，SS，ES，さらにPCのレジスタは0000_{16}にリセットされる．
3．CSレジスタは$FFFF_{16}$に設定される．

実行はメモリ位置$FFFF0_{16}$から続行される．

MN/$\overline{MX}$の影響を受ける信号には次のものがある．

マックス　ミニ
$\overline{S0}$-($\overline{DEN}$)
$\overline{S1}$-(DT/$\overline{R}$)
$\overline{S2}$-(M/$\overline{IO}$)
$\overline{RQ}/\overline{GT0}$-(HOLD)
$\overline{RQ}/\overline{GT1}$-(HLDA)
QS0-(ALE)
QS1-($\overline{INTA}$)
$\overline{LOCK}$-($\overline{WR}$)

MN/$\overline{MX}$が接地されていると，8086は“マキシマム・モード”にあると呼ばれる．MN$\overline{MX}$がハイのとき，8086は“ミニマム・モード”にあると呼ばれる．

$\overline{S0}$-($\overline{DEN}$)．MN/$\overline{MX}$ピンが接地されていると，このピンは$\overline{S0}$として機能する．$\overline{S0}$は，$\overline{S1}$-(DT/$\overline{R}$)と$\overline{S2}$-(M/$\overline{IO}$)と共にステータス情報を得るために用いられる．このステータス情報については，$\overline{S2}$-(M/$\overline{IO}$)の記述に続いて論じる．MN/$\overline{MX}$ピンのレベルがハイならば，このピンは$\overline{DEN}$として機能する．$\overline{DEN}$は，(DT/$\overline{R}$で決められる)システムあるいはローカルのバスへのバッファのデータ・トランシーバをイネーブルにすることによって，8286/8287バッファを制御するために用いられる．

$\overline{S1}$-(DT/$\overline{R}$)．MN/$\overline{MX}$ピンが接地されていると，このピンは$\overline{S1}$として機能する．$\overline{S1}$は，$\overline{S0}$-($\overline{DEN}$)と$\overline{S2}$-(M/$\overline{IO}$)と共にステータス情報を得るために用いられる．

この情報については，$\overline{\mathrm{S2}}$-(M/$\overline{\mathrm{IO}}$)の記述に続いて論じる．MN/$\overline{\mathrm{MX}}$ピンのレベルがハイならば，このピンはDT/$\overline{\mathrm{R}}$として機能する．DT/$\overline{\mathrm{R}}$は，データ伝送方向を示して，8286/8287バッファをコントロールするために用いられる．DT/$\overline{\mathrm{R}}$がハイならば，トランシーバはシステム・バスにデータを設定し，DT/$\overline{\mathrm{R}}$がローならば，トランシーバはシステム・バスからデータを取り去る．8288バス・コントローラはまた$\overline{\mathrm{DEN}}$とDT/$\overline{\mathrm{R}}$の出力を生成する．バス・コントローラが存在すれば，その$\overline{\mathrm{DEN}}$とDT/$\overline{\mathrm{R}}$の出力は，8086の$\overline{\mathrm{DEN}}$とDT/$\overline{\mathrm{R}}$の出力の代わりに用いられる．この異なる構成については後で述べる．

$\overline{\mathrm{S2}}$-(M/$\overline{\mathrm{IO}}$)． MN/$\overline{\mathrm{MX}}$ピンが接地されていると，このピンは$\overline{\mathrm{S2}}$として機能する．$\overline{\mathrm{S2}}$は，$\overline{\mathrm{S0}}$-($\overline{\mathrm{DEN}}$)と$\overline{\mathrm{S1}}$-(DT/$\overline{\mathrm{R}}$)と共に以下に述べるステータス情報を得るために用いられる．MN/$\overline{\mathrm{MX}}$ピンのレベルがハイならば，このピンはM/$\overline{\mathrm{IO}}$として機能する．メモリまたはI/Oのアクセスの間，M/$\overline{\mathrm{IO}}$はメモリ・アクセスに対してはハイに，I/Oアクセスに対してはローになる．

MN/$\overline{\mathrm{MX}}$が接地されていると，次のように$\overline{\mathrm{S0}}$，$\overline{\mathrm{S1}}$，さらに$\overline{\mathrm{S2}}$によって，8288バス・コントローラにステータスが供給される．

$\overline{\mathrm{S2}}$	$\overline{\mathrm{S1}}$	$\overline{\mathrm{S0}}$	
0	0	0	インタラプト・アクノリッジ
0	0	1	I/Oリード
0	1	0	I/Oライト
0	1	1	ホルト
1	0	0	インストラクション・フェッチ
1	0	1	メモリ・リード
1	1	0	メモリ・ライト
1	1	1	インアクティブ

この情報は，マキシム・モード・システムのためのメモリとI/Oのコントロール信号を生成するために，8288バス・コントローラによって用いられる．

QS0-(ALE)． MN/$\overline{\mathrm{MX}}$ピンが接地されていると，このピンはQS0として機能する．QS0は，QS1-($\overline{\mathrm{INTA}}$)と共に8086の命令キューの状態を得るために用いられる．詳細は後で述べるが，命令キューは8086マイクロプロセッサで6バイトの領域がある．これは，実行を待つオブジェクト・コード・バイトの保持に用いられる．MN/$\overline{\mathrm{MX}}$ピンのレベルがハイであれば，QS0-(ALE)はALEとして機能する．有効なメモリ・アドレスがアドレス/データ・バスに存在する間は，ハイのALEパルスが出力される．マキシマム・モード・システムで，ALEは8288バス・コントローラによって与えられる．

QS1-($\overline{\mathrm{INTA}}$)， MN/$\overline{\mathrm{MX}}$ピンが接地されていると，このピンはQS1として機能する．QS1は，QS0-(ALE)と共に，以下に述べるように8086の命令キューの状態を得るために用いられる．MN/$\overline{\mathrm{MX}}$ピンのレベルがハイならば，このピンは$\overline{\mathrm{INTA}}$として機能する．

8086がインタラプト・アクノリッジ・シーケンスを実行している間，$\overline{\mathrm{INTA}}$はローになる．マキシマム・システムで，INTAは8288バス・コントローラによって与えられる．MN/$\overline{\mathrm{MX}}$ピンが接地されていると，8086の命令キューの状態は，次のようにQS0とQS1によって与えられる．

QS0	QS1	
0	0	ノー・オペレーション
0	1	命令の最初のバイトが実行されている．
1	0	キューが空になっている．
1	1	命令の次のバイトがキューから取り出されている．

ＱＳ０とＱＳ１は，キュー操作に続くクロックの期間，有効となる．

$\overline{\text{RQ}}/\overline{\text{GT0}}$-(HOLD)．　ＭＮ/$\overline{\text{ＭＸ}}$ピンが接地されていると，このピンは$\overline{\text{ＲＱ}}/\overline{\text{ＧＴ０}}$として機能する．$\overline{\text{ＲＱ}}/\overline{\text{ＧＴ０}}$は要求／許可のラインである．他のバス・マスタは，このピンにローのパルスを入力することによって，8086をＨＯＬＤ状態にできる．8086は，ＨＯＬＤ状態になったことを，$\overline{\text{ＲＱ}}/\overline{\text{ＧＴ０}}$でローのパルスを出力して，要求元のバス・マスタに通知する．次いで8086は，システム・バスとスリーステート出力の制御を放棄する．新しいバス・マスタが次にシステム・バスの制御を放棄すると，もう１つのローの$\overline{\text{ＲＱ}}/\overline{\text{ＧＴ０}}$パルスを送って同様のことを行なう．そして8086はバス制御を再び確立する．要求／許可のシーケンスは８章で詳細に述べる．ＭＮ/$\overline{\text{ＭＸ}}$ピンのレベルがハイならば，$\overline{\text{ＲＱ}}/\overline{\text{ＧＴ０}}$-(ＨＯＬＤ）はＨＯＬＤとして機能する．ＨＯＬＤは，外部ロジックによってＨＯＬＤのリクエスト・ラインとして用いられる．外部ロジックがＨＯＬＤのレベルをハイにすると，8086は現在のバス・サイクルを完了してＨＯＬＤ状態になる．8086は，ＨＬＤＡにハイを出力してＨＯＬＤ状態となったことを通知する．

$\overline{\text{RQ}}/\overline{\text{GT1}}$-(HLDA)．ＭＮ/$\overline{\text{ＭＸ}}$ピンが接地されていると，このピンは$\overline{\text{RQ}}/\overline{\text{GT1}}$として機能する．$\overline{\text{RQ}}/\overline{\text{GT1}}$は，$\overline{\text{RQ}}/\overline{\text{GT0}}$よりもプライオリティが低いことを除いて，機能的には$\overline{\text{RQ}}/\overline{\text{GT0}}$と同一である．$\overline{\text{RQ}}/\overline{\text{GT0}}$の要求／許可のシーケンスが進行中でなければ，8086は$\overline{\text{RQ}}/\overline{\text{GT1}}$の要求／許可のシーケンスを開始することができる．要求／許可のシーケンスの詳細は８章に述べられている．ＭＮ/$\overline{\text{ＭＸ}}$ピンのレベルがハイならば，$\overline{\text{RQ}}/\overline{\text{GT1}}$-(HLDA）はＨＬＤＡとして機能する．ＨＬＤＡはＨＯＬＤアクノリッジの信号である．HLDAは，ＨＯＬＤによって作られたホールド要求を確認するために，ハイが出力される．ＨＬＤＡの信号がハイになると，8086ＣＰＵはまた，そのスリーステートの出力信号をフロート状態にする．したがって，システム・バスをフロート状態とする．

$\overline{\text{LOCK}}$-($\overline{\text{WR}}$)．ＭＮ/$\overline{\text{ＭＸ}}$ピンが接地されていると，このピンは$\overline{\text{ＬＯＣＫ}}$として機能する．$\overline{\text{ＬＯＣＫ}}$は，命令実行中に8086がシステム・バスの制御を失なうことを避けるために，ローが出力される．$\overline{\text{ＬＯＣＫ}}$がローの間，外部ハードウエアは，他のバス・マスタがシステム・バスの制御を獲得しないことを保証しなければならない．8086がＬＯＣＫ命令を実行すると，次の命令が実行される間，$\overline{\text{ＬＯＣＫ}}$はローが出力される．ＭＮ／$\overline{\text{ＭＸ}}$ピンのレベルがハイならば，$\overline{\text{ＬＯＣＫ}}$-($\overline{\text{ＷＲ}}$）は$\overline{\text{ＷＲ}}$として機能する．$\overline{\text{ＷＲ}}$は，メモリまたはI/Oのライトの間，ローが出力される．アドレス／データ・バスに出力されているデータが安定なときに，パルスの立下りエッジが生じる．

7.1.3　パワーとタイミングのライン

ＣＬＫは，すべての8086ロジックを同期させるために用いられるクロック信号である．

この信号は普通，8284クロック・ジェネレータによって出力される．

Vcc はパワー・サプライのピンである．8086は，このピンに＋５Ｖ±10％が加えられる必要がある．

２つのＧＮＤピンが存在し，これらは共に接地のピンである．

7.1.4 スリーステートのラインと信号

次の8086の信号はスリーステートである．

AD0 - AD15
A16/S3
A17/S4
A18/S5
A19/S6
$\overline{\mathrm{BHE}}$/S7
$\overline{\mathrm{RD}}$
$\overline{\mathrm{S0}}$-($\overline{\mathrm{DEN}}$)
$\overline{\mathrm{S1}}$-(DT/$\overline{\mathrm{R}}$)
$\overline{\mathrm{S2}}$-(M/$\overline{\mathrm{IO}}$)
$\overline{\mathrm{LOCK}}$-($\overline{\mathrm{WR}}$)
$\overline{\mathrm{INTA}}$

8086がＨＯＬＤ状態にある間，これらすべての信号はハイ・インピーダンス状態になる．$\overline{\mathrm{S0}}$-($\overline{\mathrm{DEN}}$)，$\overline{\mathrm{S1}}$-(DT/$\overline{\mathrm{R}}$)，$\overline{\mathrm{S2}}$-(M/$\overline{\mathrm{IO}}$) の信号は，8086がホールド・アクノリッジを出す直前に，フロート状態になる．

インタラプト・アクノリッジの間，AD0 - AD15，A16/S3，A17/S4，A18/S5，A19／Ｓ６のラインはフロート状態になる．

7.2 8086の概要と基本的システムの概念

この節では，バス・サイクル，アドレス／データ・バス，システム・データ・バス，エグゼキューション・ユニット，バス・インターフェイス・ユニット，命令キューを含む基本的な8086のシステムの概念について論じる．

7.2.1 8086バス・サイクルの定義

8086は，システム・バスを通して外部ロジックと連絡する．8086は，データを伝送または命令をフェッチするために，“バス・サイクル”を実行する．“バス・サイクル”を図7-2 に示す．

最小のバス・サイクルは，Ｔステートと呼ばれる４つのＣＰＵクロックの期間より成る．最初のＴステート（Ｔ１）の間，8086は，20ビットの多重化アドレス／データ／ステータス・バスにアドレスを出力する．バス上のアドレスは，ＡＬＥ信号がハイからローに遷移するとき，有効とみなされる．ミニマム・システムでは，この信号は8086によって作られる．マキシマム・システムでは，この信号は 8288 バス・コントローラによって作られる．

$\overline{S2}$-(M/$\overline{IO}$) 信号は，メモリまたはI/Oのアクセスのどちらが実行されているかを示す．

第2のＴステート（Ｔ2）の間，8086はアドレス・バスからアドレスを取り去る．リード・バス・サイクルに対しては，リード・サイクルの準備としてデータ・バス・ラインはフロート状態になる．ライト・バス・サイクルに対しては，データ・バス・ラインにデータが出力される．

データ・バス・トランシーバは，8086のシステム構成と伝送の方向（8086へか8086からか）に依存して，Ｔ1またはＴ2の間，イネーブルとなる．読み出し，書き込み，またはインタラプト・アクノリッジのコントロール信号はＴ2の間，常に有効である．

Ｔ2の間，バス・サイクル・ステータス（Ｓ3，Ｓ4，Ｓ5，Ｓ6）は，上位4つのアドレス／ステータス・バス・ラインに出力される．次に示すステータス情報は，残りのバス・サイクルで有効である．

S4	S3	
0	0	エキストラ(ES セグメントに対する相対)
0	1	スタック　(SS セグメントに対する相対)
1	0	コード/なし(CS セグメントに対する相対またはデフォルト・ゼロ)
1	1	デ　ー　タ(DS セグメントに対する相対)
S5＝IF　(インタラプト・イネーブル・フラグ)		
S6＝0　(8086がバスを使用していることを示す)		

8086は，Ｔ3の間，上位4つのアドレス／ステータス・バス・ラインに，ステータス情報を出し続ける．また，8086はライト・バス・サイクルの間，データを出力し続ける．リード・バス・サイクルに対して，8086はＴ3の終わりにデータの入力を行なう．選択された素子に必要な速度でデータの伝送を行なう能力がなければ，素子はREADYにローを入力して“ノット・レディ”を知らせる必要がある．

これにより，8086はＴ3の後に付加的なクロック・サイクルを挿入する．この付加的なクロック・サイクルは，Ｔwステート（ウエート・ステート）と呼ばれる．“ノット・レディ”の表示は，Ｔ3の開始前にＣＰＵに示されなければならない．Ｔwの間のバス動作はＴ3と同じである．選ばれた素子が伝送完了の十分な時間を有していれば，READYをハイにする．Ｔwクロック期間が終わった後で，Ｔ4，すなわちバス・サイクルの最後のクロック期間が実行される．

Ｔ4の間に，メモリとＩ/Ｏのコントロール・ラインはディスエーブルとなり，選ばれた外部素子はシステム・バスから切り離される．

図7.2 に基本的な8086バス・サイクルを示す．

バス・サイクルは，システム・バス上の素子に対して，素子と素子内のレジスタまたはメモリ位置を選択するためのアドレス，さらにデータに伴うリード・ストローブまたはライト・ストローブより成る非同期の事象に見える．選択された素子は，ライト・サイクルの間にデータを受け取り，リード・サイクルの間にバス上にデータを設定しなければなら

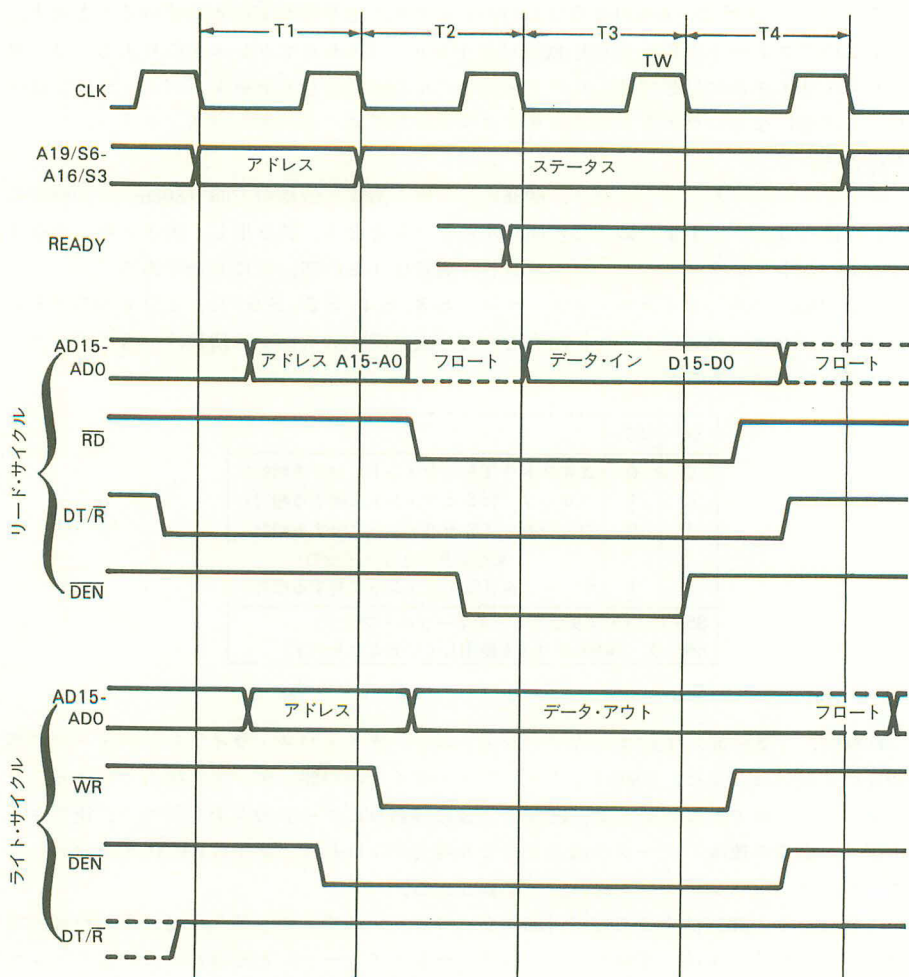



図7-2 基本的な8086のバス・サイクル

ない．バス・サイクルの終わりに，素子は書かれるデータをラッチまたはバス・ドライバをディスエーブルにする．素子がバス・サイクルを変更できるただ１つの方法は，READYのコントロール・ラインによってウエート状態のクロック期間を挿入することで行なえる．

命令のフェッチが行なわれているか，あるいは8086とメモリまたはI/O素子の間でオペランドが伝送されるべきときは，8086は単にバス・サイクルを実行するだけである．バス・サイクルが実行されていないときは，8086のバス・インターフェイス・ロジックはアイドル・クロック期間（ＴＩと呼ぶ）を実行する．アイドル・クロックの期間，8086は以前のバスサイクルから上位４つのアドレス・ラインにステータス情報を出力し続ける．以前のバス・サイクルが書き込みならば，次のバス・サイクルの開始までＣＰＵは16個のデータ・バス・ラインにデータを出力し続ける．8086がリード・サイクルに続いてアイドル・ク

ロック期間を実行するならば，次のバス・サイクルの開始まで8086は16個のデータ・バス・ラインをフロート状態にする．

メモリのアクセスに際して，8086は２つのタイプの操作を行なう．

- 命令のフェッチ
- 命令に必要なオペランドの読み込みまたは書き込みを行なうためのメモリ・アクセス

命令フェッチとメモリ・アクセスのバス・サイクルの間の，通常簡単な連続した関係は，8086内では６バイトの命令オブジェクト・コードのキューの存在によって変えられている．

8086のバス・インターフェイス・ロジックがアイドルでなければ，命令フェッチのバス・サイクルを実行する代わりに，命令キューが満たされるまで，プログラム・メモリから次に続くオブジェクト・コード・バイトをフェッチする．１つ以上の命令のオブジェクト・コードがキューにあれば，命令フェッチのバス・サイクルは，付加的な命令フェッチのバス・サイクルによってオペランドのメモリ・アクセス・バス・サイクルと分離される．

ジャンプまたはコールの命令が実行されると，現在の命令キューにある次に続く命令のオブジェクト・コード・バイトはもはや必要でなくなる．したがって，キューの内容は何の悪影響もなく捨てられる．

7.2.2　8086のアドレスとデータ・バスの概念

8086に接続されるほとんどのメモリ素子と周辺装置は，全バス・サイクルの間，安定なアドレスを必要とするので，Ｔ１の間に多重化されたアドレス／データ・バス上に存在するアドレスはラッチされなければならない．このラッチされたアドレスは，必要な周辺装置またはメモリ位置を選択するために用いられる．アドレス／データ・バスの分離のために，8086にはアドレス・ラッチ・イネーブル信号（ＡＬＥ）があり，これは8282あるいは8283の８ビット双安定ラッチにアドレスを捕えるために用いることができる．

このラッチはインバート（8283）またはノンインバート（8282）であり，32ｍＡのドライブ能力を持ち，22 ns（インバート）または30 ns（ノンインバート）で300 pF の容量負荷をスイッチできるトライステート・バッファによって駆動される出力を有する．8282/8283ラッチは，ＡＬＥがハイの間，アドレスを出力に伝え，ＡＬＥの次のエッジでアドレスをラッチする．しかしこれは，ラッチの伝播遅延で，アドレス・アクセスとチップ・セレクトのデコードを遅らせる．

図7.3　にアドレス／データ・バスの分離を示す．

ラッチの出力は，ロー・アクティブの$\overline{\text{ＯＥ}}$入力によってイネーブルとなる．多重化アドレス/データ・バスの分離（多重化バスからのアドレスのラッチ）は，単独のアドレス・バスがシステム全体にアドレスを分配して，システムの適当な時点またはＣＰＵで局所的に処理できる．

最適なシステムの性能と，マルチプロセッサと Multibus 構成との互換性から，**図7-4**に示す局所的信号分離は，**図7-5**に示されている分散信号分離よりも強く推薦される．この章の残りでは，図7-4で示しているように，バスはCPU側で分離されることを仮定している．

8086のメモリ・アドレス領域は，バイトが８ビットのデータ要素を含み，２つの連続す

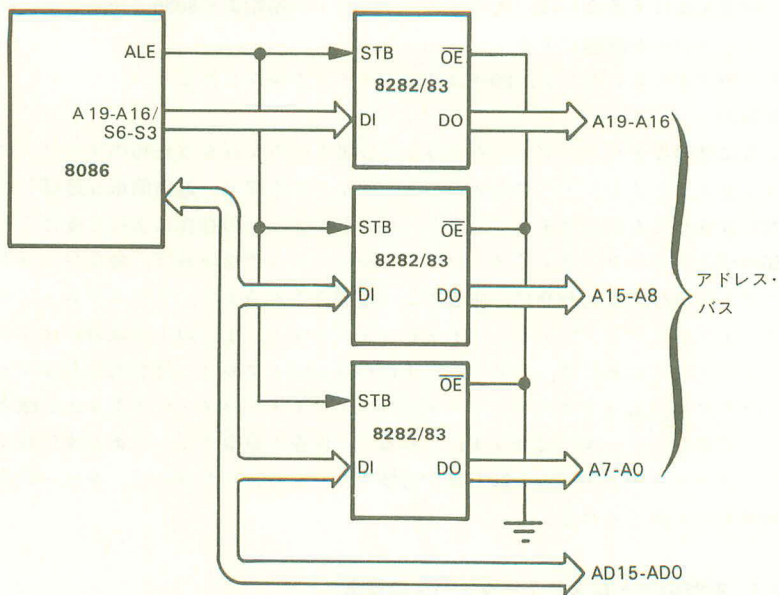



図7-3　アドレス／データ・バスの分離

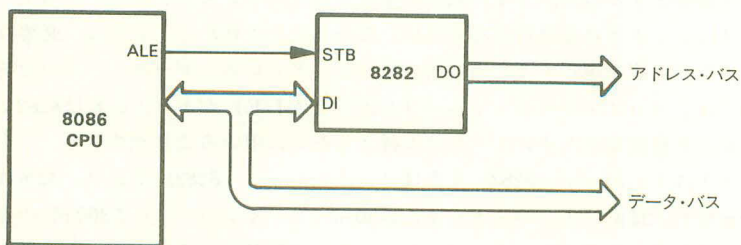



図7-4　別々のアドレスとデータのバス

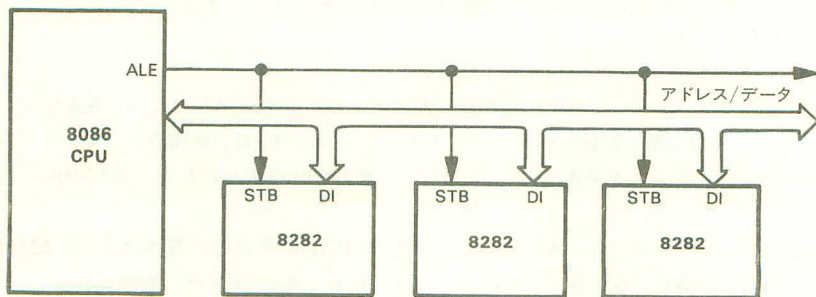
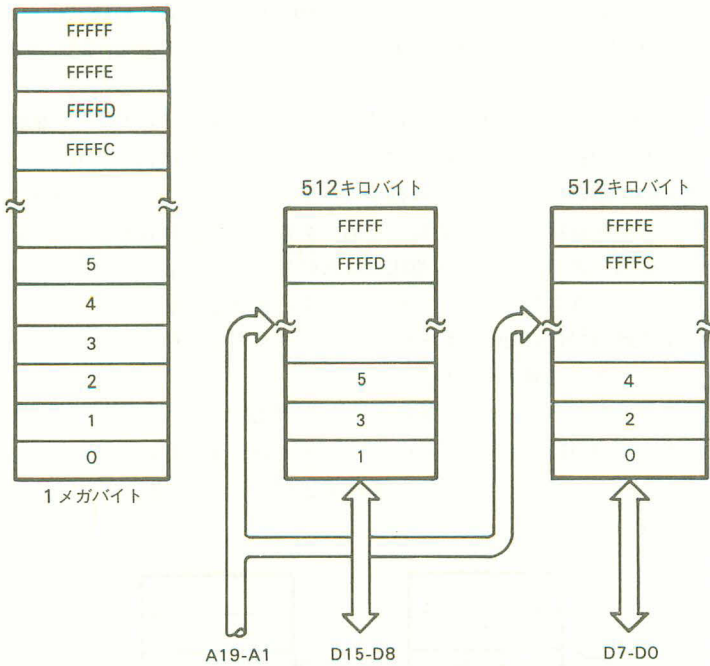
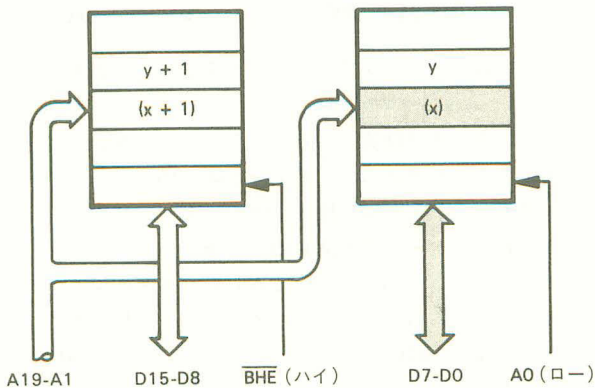
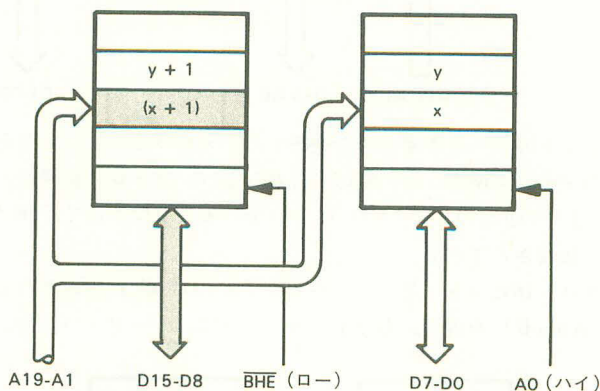



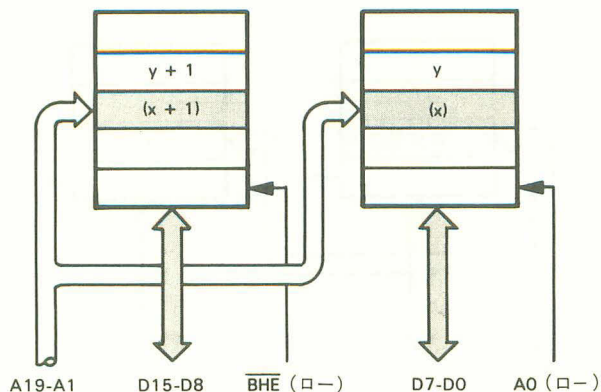
図7-5　局所的アドレスの分離の多重化バス

るバイトが16ビットのデータ要素を含む連続した1メガバイトとして考えられる．バイトまたはワードのアドレス境界に何の制限も存在しない．アドレス領域は，この領域を各々512キロバイトまでの2つの8ビットのバンクに分けることによって，16ビットのデータ・バスに物理的に接続されている．

1つのバンクは，16ビット・データ・バスの下位（D7-0）に接続されていて，偶数アドレス・バイト（A0＝0）を含む．もう1つのバンクは，データ・バスの上位（D15-8）に

接続されていて，奇数アドレス・バイトを含む(A0=1)．各バンクの特定のバイトは，アドレス・ラインA19-A1によって選択される．データ・バスの下位（D7-0）を通して偶数アドレスにデータ・バイトは伝送される．

A0はローのとき，データ・バスの下位に接続されているメモリ・バンクを選択し，バス・ハイ・イネーブル($\overline{BHE}$）はハイを出力してデータ・バスの上位のメモリ・バンクをディスエーブルにする．このディスエーブル操作は，下位メモリ・バンクへの書き込み操作で上位バンクのデータを破壊することを防ぐために必要である．$\overline{BHE}$もまた，A19-A16アドレス・ラインと同じタイミングで多重化された信号なので，全バス・サイクルの間，安定な信号を得るためにALEによってラッチされなければならない．

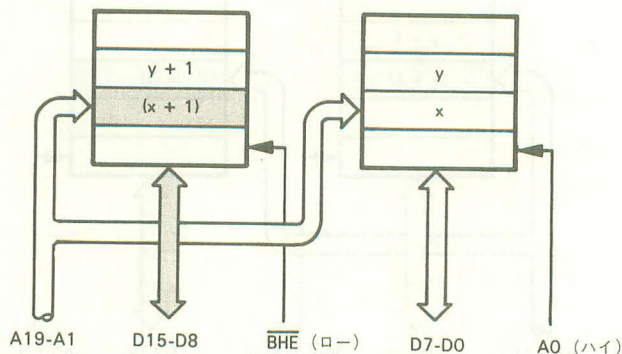
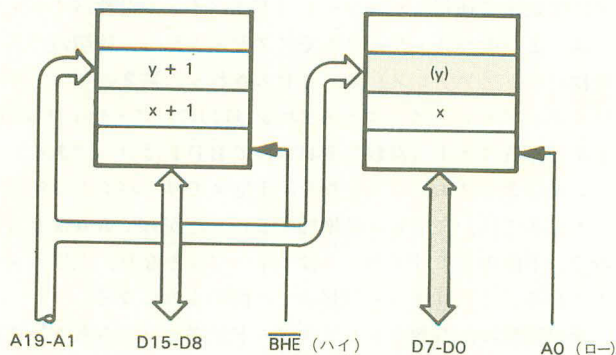
T2からT4の間，$\overline{BHE}$の出力はステータス・ラインS7で利用できる（ステータス・ラインS7の意味はまだ定義されていない)．

奇数アドレスのメモリ・バイトにアクセスすると，情報はデータ・バスの上位(D15-D8)を通して伝送される．$\overline{BHE}$は上位メモリ・バンクを有効にするためにローが出力される．A0は下位メモリ・バンクをディスエーブルにするためにハイが出力される．これは次のように図示される．

8086は，データ・バスの適正な片方を通してデータを伝送し，必要な信号レベルの$\overline{BHE}$とA0を出力する．

例として，奇数アドレスのメモリ位置からCLレジスタ（CXレジスタの下位）に1バイトのデータをロードすることを考える．このデータは，16ビット・データ・バスの上位を通してアクセスされる．このデータは上位8つのデータ・バス・ラインによって8086に伝送されるが，8086は自動的にデータの方向を内部の16ビット・データ・バスの下位，したがってCLレジスタに変える．この能力により，ALレジスタによるバイトのI/O伝送の，16ビット・データ・バスの上位あるいは下位に接続されているI/O素子へのアクセスが可能となる．偶数アドレスに位置する16ビットのワード(下位バイトが偶数バイト・アドレスの連続した2バイト)は，1つのバス・サイクルでアクセスされる．A19-A1は各バンクの適当なバイトを選択し，A0のローと$\overline{BHE}$のローによって両方のバンクが同時にイネーブ

ルとなる.

奇数アドレスに位置する16ビットのワード（下位バイトが奇数バイト・アドレスの連続した2バイト）は，2つのバス・サイクルでアクセスされる．最初のバス・サイクルで，下位バイト（奇数バイト・アドレス）がアクセスされる．第2のバス・サイクルで，上位バイト（偶数バイト・アドレス)がアクセスされる．最初のバス・サイクルの間，A19- A1はアドレスを指定する．A0は1（奇数アドレス）で$\overline{BHE}$はローである．したがって，下位のメモリ・バンクはディスエーブルとなり，上位メモリ・バンクはイネーブルとなる．第2のバス・サイクルでは，アドレスは増加する．したがってA0は0となる．しかし$\overline{BHE}$はハイなので，下位のメモリ・バンクはイネーブルとなり，上位のメモリ・バンクはディスエーブルとなる．これは次ページに示した図のようになる．

上に示した一連の過程は，奇数アドレスのワード伝送が指定されたときは常に，8086によって自動的に実行される．8086は，内部の16ビット・レジスタの上位と下位のバイトと，データ・バスの適当な部分とを自動的に接続する．ただし，奇数アドレス境界のワードにアクセスするには余分のバス・サイクルを必要とすることに注意．これはシステムの性能を低下させる．

バイト・リードの期間，データ・バスの上位または下位で両方ではない位置にデータが予期されても，CPUはクロック期間T2の間，16ビット・データ・バスのすべてをフロート状態にする．これにより，読み出し専用素子（ROM，EPROM）に対するチップ・セレクトのデコードの条件が簡単になる（チップ・セレクトのロジックについては後で述べる）.バイト・ライト操作の間，8086は16ビット・データ・バスのすべてを駆動する．データが伝送されていない方のデータ・バスの情報は不定である．このような概念はI/Oアドレス領域にも適用される．

7.2.3 システム・データ・バスの概念

システム・データ・バスを取り上げると，(a)図7-6に示す多重化アドレス/データ・バス，あるいは(b)図7-7（374ページ）に示すような，トランシーバによって多重化バスにバッファを用いたデータ・バスの2つの実現のどちらかを考えなければならない．

多重化データ・バスを用いるとき，設計者は多重化バスに直接接続されるメモリまたはI/O素子がT1の期間にバス上のアドレスを壊さないことを保証しなければならない．この情況を避けるために，図7-8に示されているように，素子の出力ドライバは，素子チップ・セレクトによってイネーブルとなってはならず，システムのリード信号によって出力のイネーブルが制御されるものでなければならない．

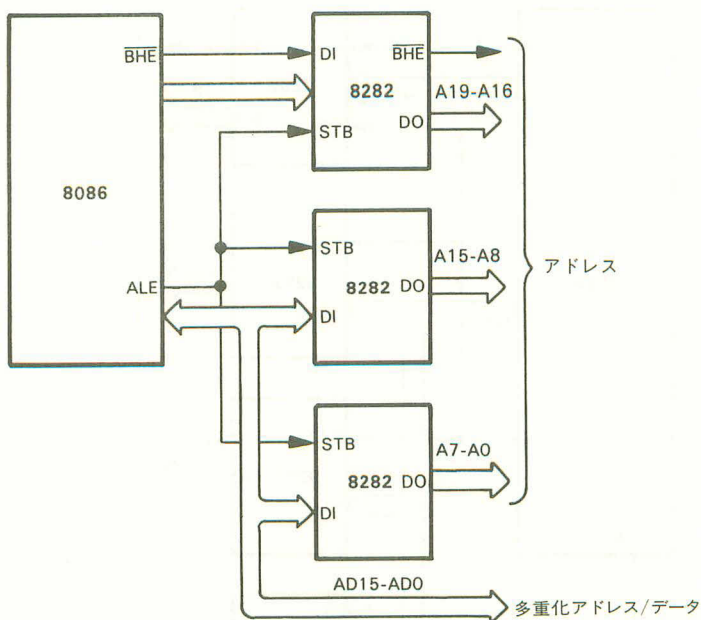



図7-6　多重化データ・バス

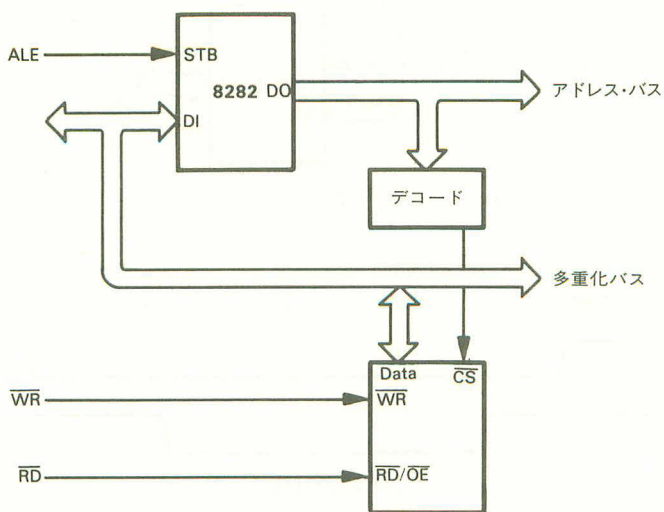



図7-8　多重化バス上のアウトプット・イネーブルを持つ素子

8086
BHE
ALE
DT/R
DEN
DI BHE
8282
A19-A16
STB DO
STB
A15-A8
8282 DO
DI
STB
A7-A0
8282 DO
DI
アドレス
OE
T
B D15-D8
8286/8287
A
データ
OE
T
B D7-D0
8286/8287
A

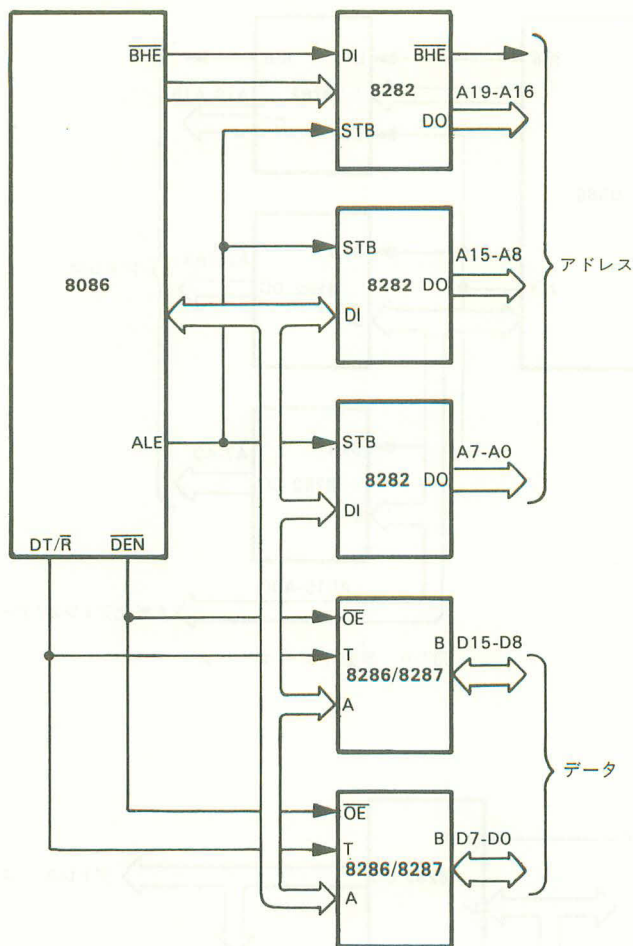
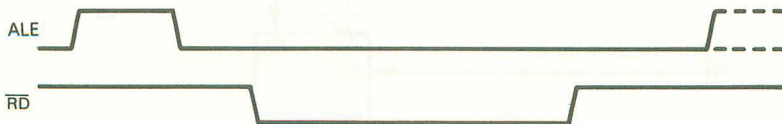
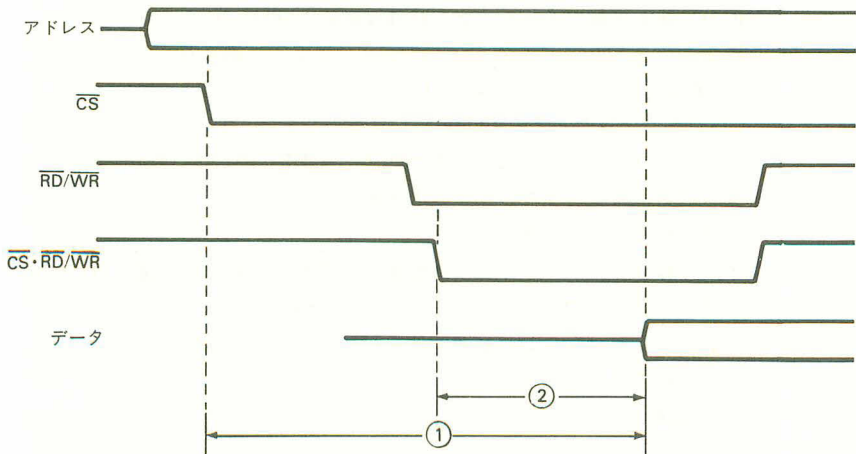



図7-7　バッファを用いたデータ・バス

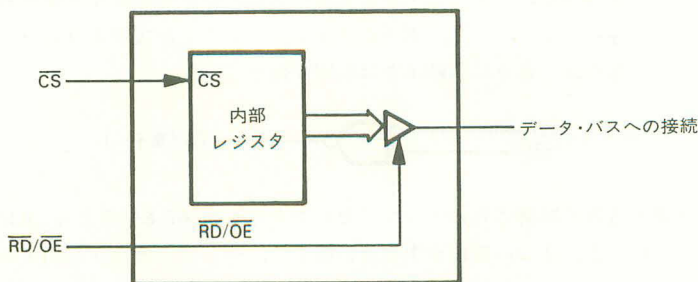
8086のタイミングは，アドレスがALEによってラッチされて多重化アドレス/データ・バスがフロート状態になるまで，読み出しは有効でないことを保証している．

（すべてのインテルのペリフェラル，EPROM製品とRAMは，マイクロプロセッサ向けのものであれば，図7-8 に示しているように多重化バスに接続させるための出力イネーブルまたはリード入力を備えている）．

出力イネーブルを持たないとき，素子を多重化バスにインターフェイスさせるために，いくつかの方法が存在する．しかし，素子のチップ・セレクトが次のようにコマンドで外部から制御されるならば，各々に制約または制限が生じる．

読み込みと書き込みで制御されるチップ・セレクトをもつ図7-8を考える．上に示した外部制御が用いられると，2つの問題が生じる．第1に，チップ・セレクトのアクセス・タイムは読み込みのアクセス・タイムにまで落ち，最大のシステム性能（ウエート・ステートがない）が達せられることを仮定すると．他の不必要に高速な補助素子を必要とさせる．これを図7-9に示す．

① アドレス・デコードで生成された$\overline{\mathrm{CS}}$に対するアクセス・タイム

② $\overline{\mathrm{CS}}$が$\overline{\mathrm{RD}}/\overline{\mathrm{WR}}$で制御された場合のアクセス・タイム

図7-9　$\overline{\mathrm{RD}}/\overline{\mathrm{WR}}$で制御された$\overline{\mathrm{CS}}$

図7-8で，分離した$\overline{\mathrm{CS}}$と$\overline{\mathrm{RD}}/\overline{\mathrm{OE}}$の入力を持つ素子では，実際にアクセスは$\overline{\mathrm{CS}}$から内部で開始するが，バスへの出力ドライバは，$\overline{\mathrm{RD}}/\overline{\mathrm{OE}}$までイネーブルとはならない．したがって，アクセス・タイムは$\overline{\mathrm{RD}}/\overline{\mathrm{OE}}$からではない．これは次のように図示される．

設計者はまた，チップ・セレクトが外部で制御されるとき，素子について，書き込みのセットアップとホールドの時間に対するチップ・セレクトが違反しないことを確かめなければならない．これを**図7-10**に示す．

別の素子選択の方法も可能であるが，また特殊な制約が課せられる．したがって，出力イネーブルを持つ素子を多重化データ・バスに接続することを勧める．

図7-6に示した多重化データ・バスのもう1つの制限は，特定のAC特性を保証する，8086の2.0mAのドライブ能力とその100pFの容量負荷である．1つのI/O素子につき，20pF，1つのアドレス・ラッチにつき12pF，1つのメモリ素子につき5-12pFの容量負荷を仮定すると，3つの周辺装置と2ないし4個のメモリ素子（各バス・ラインにつき）を合わせたシステムは負荷の制限に非常に近い．

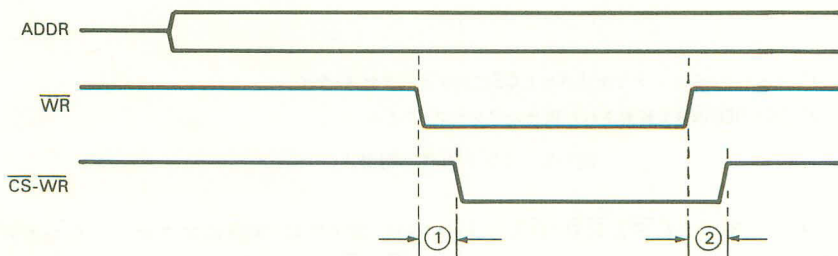
大きいシステムの容量負荷とドライブの必要条件を満たすためには，**図7-11**に示すようにデータ・バスにバッファを用いなければならない．8286ノンインバートと8287インバートのオクタル・トランシーバは，この必要条件を満たすために，8086ファミリーの一部として提供される．これらは，バス・インターフェイスで32mA，コンポーネント・インターフェイスで10mAを駆動するバッファを持ち，さらに，22ns（8287）または30ns（8286）で，

① $\overline{\text{CS}}$は，書き込みより前に有効ではなく，1または2つのゲート遅延の後にアクティブとなる．
② $\overline{\text{CS}}$は書き込みの後も1または2つのゲート遅延の間，有効となっている．

図7-10　$\overline{\text{WR}}$のセットアップとホールドに対する$\overline{\text{CS}}$

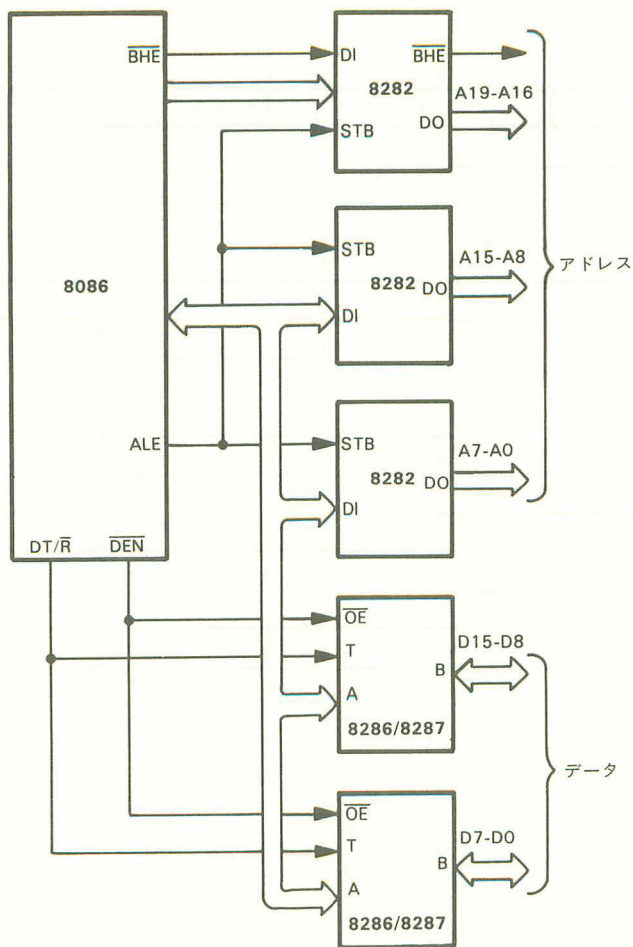
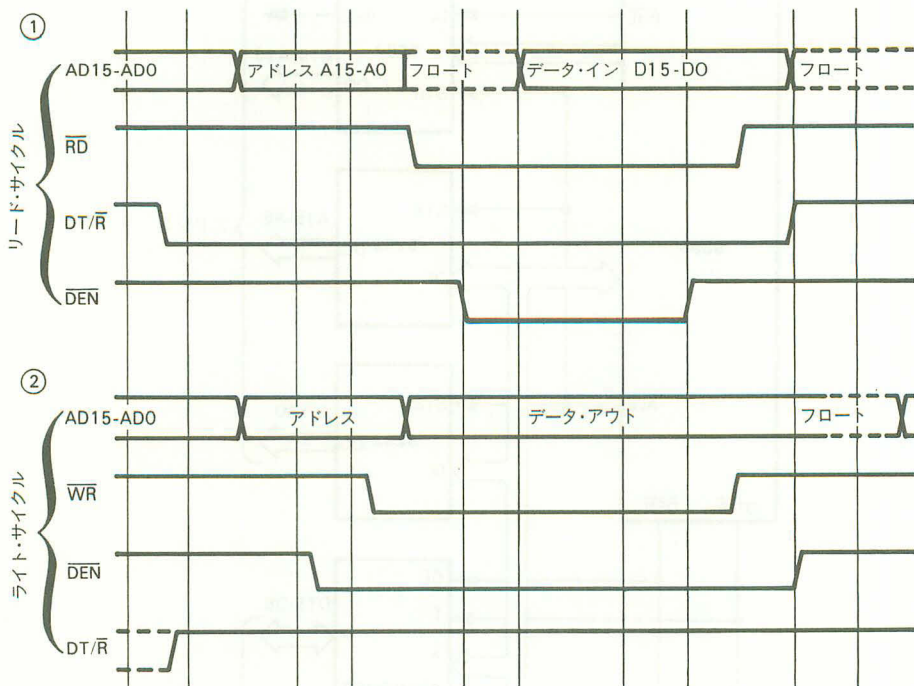



図7-11　バッファを用いたデータ・バス

バス・インターフェイスで300pF，コンポーネント・インターフェイスで100pFの容量負荷のスイッチが可能である．8086システムは，図7-12に示されているように，8286と8287のトランシーバをイネーブルにしてその方向を制御するための，データ・イネーブル（$\overline{\text{DEN}}$）とデータ・トランスミット／レシーブ（DT／$\overline{\text{R}}$）の信号を備えている．

これらの信号は，T1の間，システムから多重化バスの分離を保証し，読み出しと書き込みのバス・サイクルの間，CPUとのバス競合を除去する適当なタイミングを有する．

メモリと周辺の素子がCPUから分離されていても，メモリ／周辺の素子がチップ・セレクトに加えて出力のイネーブル制御を有していなければ，なおシステムにはバス競合が存在する．

① 8086が多重化バスをフロート状態とした後に$\overline{DEN}$はイネーブルになる．

② $\overline{DEN}$はサイクルの初期にトランシーバをイネーブルにするが，DT/$\overline{R}$はトランシーバが受信ではなく送信のモードにあり，CPUに対する駆動を行なわないことを保証する．

図7-12　バス・トランシーバの制御

この構成を**図7-13**に示す．たとえば，1つのチップ・セレクトから他へ遷移する間にバス競合が起き，新しく選択された素子は，以前に選択された素子がそのドライバをディスエーブルにする前に，バスの駆動を開始する．より重要な問題はライト・バス・サイクルの間に生じる．出力がチップ・セレクトだけで制御されている素子は，CPUによってトランシーバを通してデータ出力がライトされると同時に，チップ・セレクトからライト・アクティブまでバスを駆動する．この状態を**図7-14**に示す．

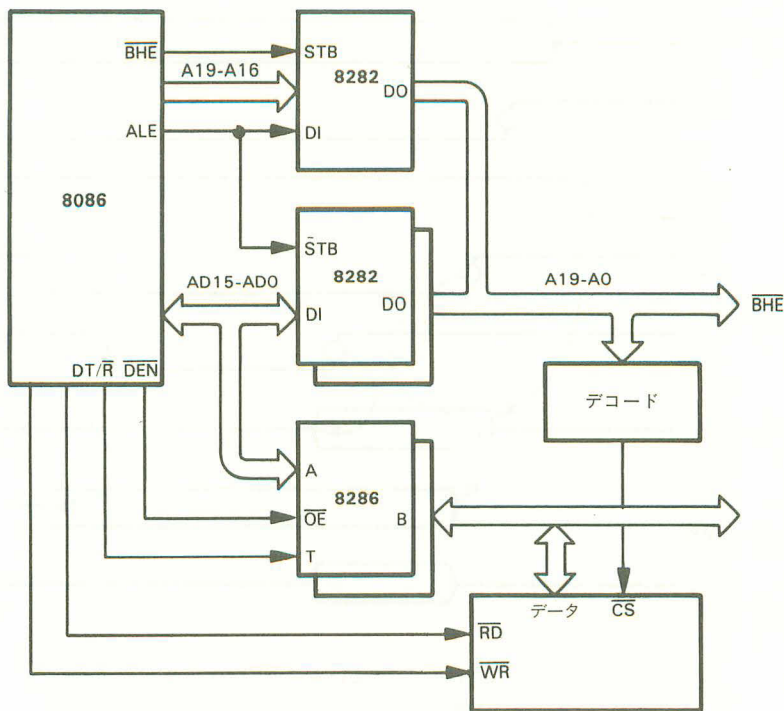



図7-13 システム・バス上のアウトプット・イネーブルを有する素子

多重化バスの周囲の選択タイミング問題に与えられた同じ方法をここで適用できるが，同じ制限を伴う．

第2レベルのバッファによって，システム・バス上で素子から見た全体の負荷を小さくできる．これを**図7-15**に示す．

一般に，二重バッファは，メモリ・アレイを分離するためにマルチボード・システムで用いられる．しかし二重バッファは，さらにアクセスの遅延を生じ，さらに重要なことに，システム・バスとそれとのインターフェイスを持つ素子への関係で，第2のトランシーバの制御に注意を払わなければならない．第2のトランシーバの制御にはいくつかの方法が利用できる．

図7-16に示す最初の方法は，単にシステム全体に$\overline{\text{DEN}}$とDT/$\overline{\text{R}}$を分配する．DT/$\overline{\text{R}}$は，第2レベルのトランシーバの適正な方向制御を行なうために，インバートされる．

図7-17に示される第2の方法は，出力イネーブルを持つ素子の制御を与える．

$\overline{\text{RD}}$は通常，周辺装置からシステム・バスへのデータの方向づけに用いられる．図7-17において，ローカル・バス上の素子が選択されたときは常にバッファが選択される．読み込みは同時に素子の出力をイネーブルとしてトランシーバの方向を変えるので，読み込みの間に素子のローカル・バスでバス競合の可能性がある．競合は，読み込みが終結する間でも起きる．

ADDR

$\overline{CS}$

DT/$\overline{R}$

$\overline{DEN}$

$\overline{WR}$

素子がバスを駆動

トランシーバがバスを駆動

バス競合（両方がバスを駆動）

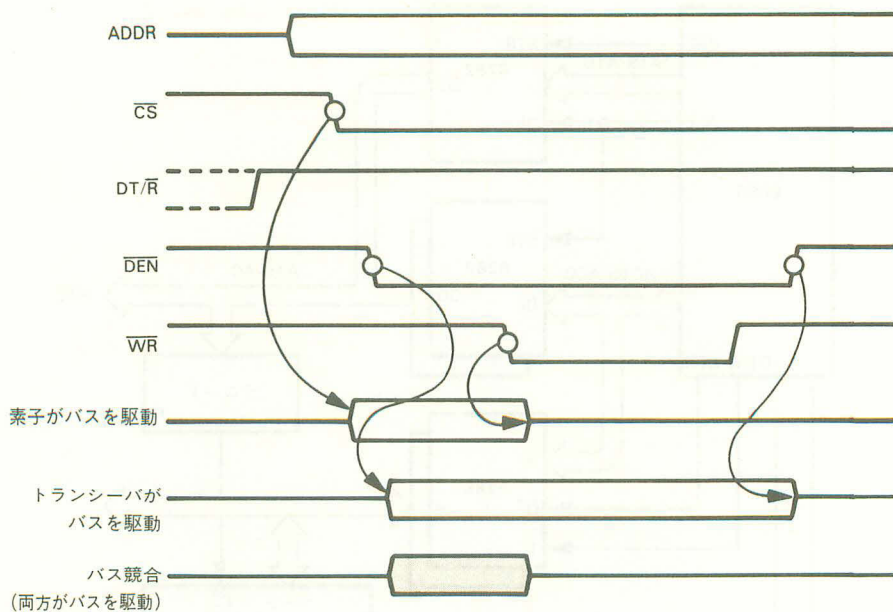



図7-14　アウトプット・イネーブルのない素子のライト期間のシステム・バスにおけるバス競合

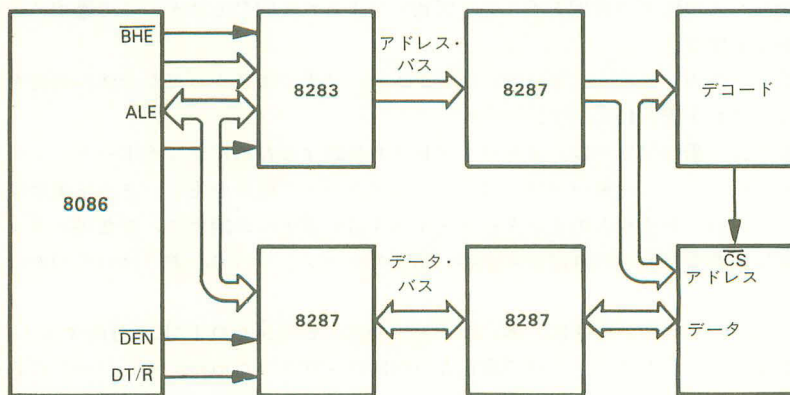



図7-15　全体にバッファを用いたシステム

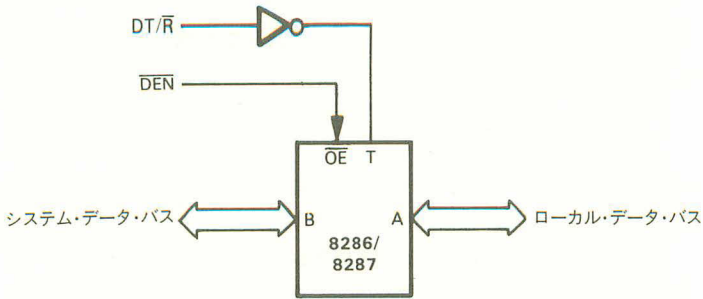



図7-16　ＤＥＮとＤＴ／Ｒによるシステム・トランシーバの制御

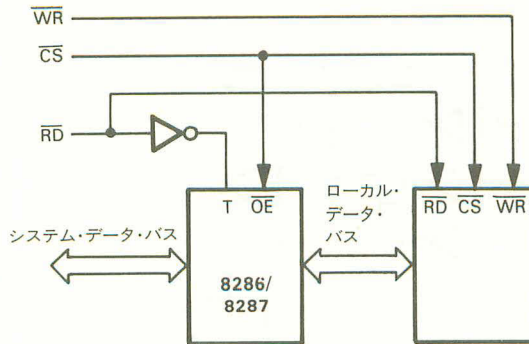



図7-17　ＯＥを有する素子

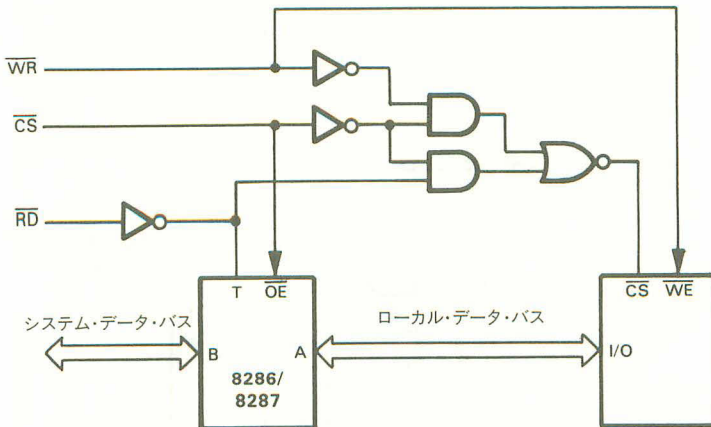



図7-18　ＯＥのない素子．共通または分離の入力／出力制限付きリード・
アクセス．ＷＥのホールドとセットアップに対して制限されたＣＳ

出力イネーブルのない素子には，素子のチップ・セレクトが読み込みまたは書き込みで決まるならば，**図7-17**に示す方法が適用できる．これを**図7-18**に示す．

読み込み／書き込みでチップ・セレクトを制御することにより，コマンドが受け付けられる前に素子が第２のトランシーバに対して駆動することを避けられる．この方法での制限を次に示す．

1．以前述べたように，アクセスは読み込み／書き込みの時間に制限される．

2．チップ・セレクトは，書き込みのセットアップとホールドの時間に制限される．

出力イネーブルのある素子とない素子に適用される別の方法を**図7-19**に示す．

$\overline{\mathrm{RD}}$は再び第２のトランシーバの方向を制限するが，コマンドとチップ・セレクトがアクティブとなるまで，イネーブルではない．バス競合の可能性はなお存在するが，トランシーバの方向変更時間に対する出力イネーブルの変動にまで縮小されている．チップ・セレクトからのすべてのアクセス・タイムが利用できる．ただし，データは書き込みより前で

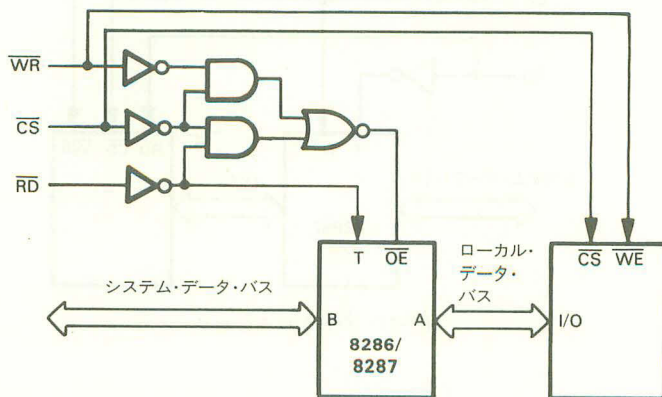



図7-19　$\overline{\mathrm{OE}}$ のない素子．共通または分離の入力／出力完全リード・アクセス．制限付きのライト・データのセット・アップとホールド

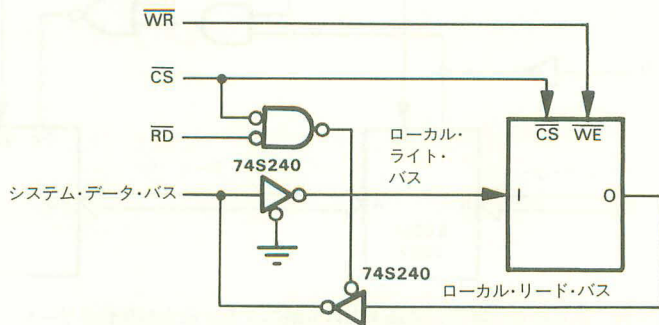



図7-20　$\overline{\mathrm{OE}}$ のない素子．分離の入力／出力

は，有効でなく，書き込みの後のトランシーバをディスエーブルにするのに必要な遅延量だけ有効として保持されているだけとなる．

最後の1つの方法は，分離した入力と出力を持つ素子のためのものである．**図7-20**を見られたい．

今までに示した1個のトランシーバではなくて，分離したバスのレシーバとドライバが用いられている．レシーバは常にイネーブルであるが，一方バス・ドライバは，$\overline{RD}$とチップ・セレクトによって制御されている．このシステムでバス競合の起きる可能性は，チップ選択が変化する間に，リード・バスの各ライン上の複数の素子がイネーブルとなってディスエーブルとなるときだけにある．

8086のインターフェイスについてはこの節を通して，多重化バスは“ローカル”なＣＰＵバス，分離されたアドレスとバッファを用いたデータのバスは“システム・バス”と考えている．

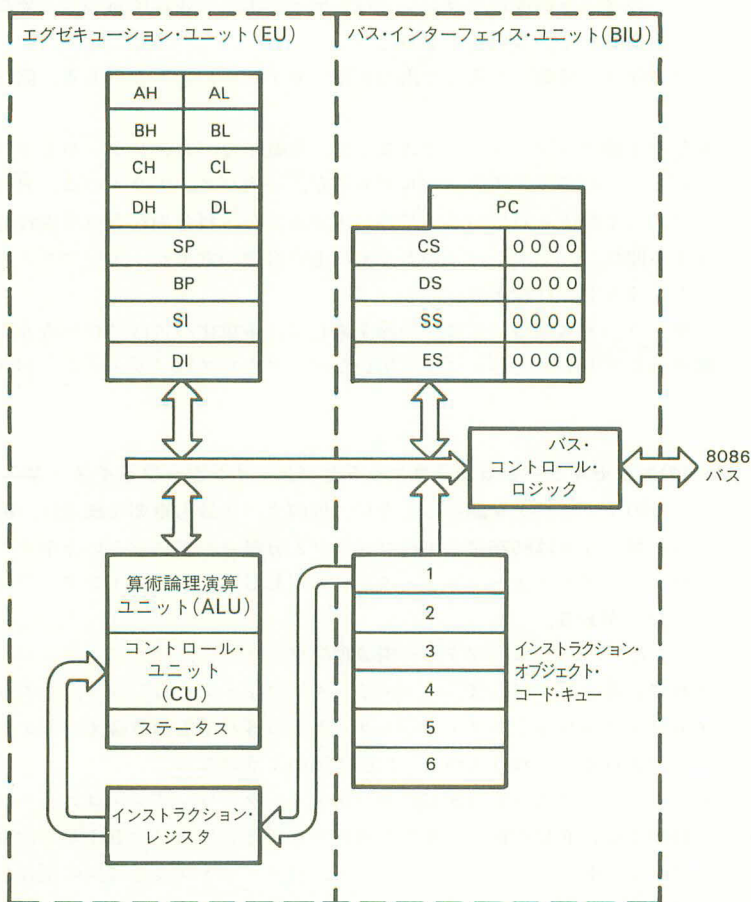
7.2.4　8086のエグゼキューション・ユニットとバス・インターフェイス・ユニット

8086の命令実行のタイミングを調べるときに，理解すべき最も重要な概念は，8086バス・コントロール・ロジックは8086命令実行ロジックと分離されているという事実である．すなわち，8086にはエグゼキューション・ユニット（ＥＵ）とバス・インターフェイス・ユニット（ＢＩＵ）がある．

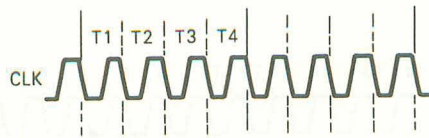
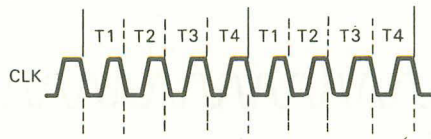
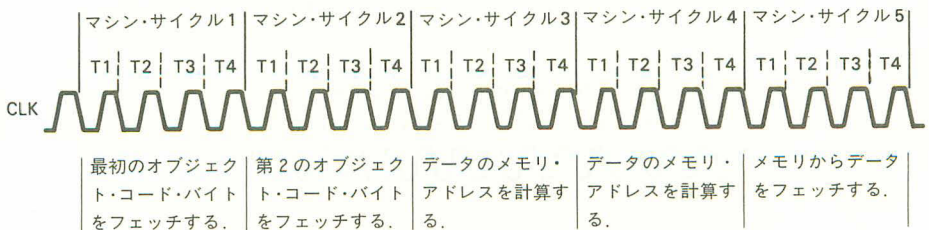
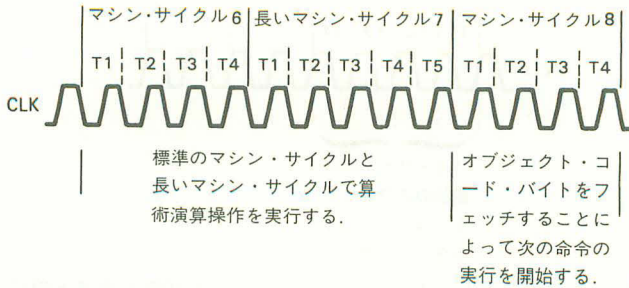
EUには，データとアドレスのレジスタ，算術論理演算ユニット，さらにコントロール・ユニットが含まれている．ＢＩＵには，バス・インターフェイス・ロジック，セグメント・レジスタ，メモリ・アドレッシング・ロジック，さらに6バイトの命令オブジェクト・コード・キューが含まれる．これは次ページに示す図のようになる．

エグゼキューション・ユニット（ＥＵ）とバス・インターフェイス・ユニット(ＢＩＵ)とは非同期に動作する．ＥＵが新しい命令を実行できるときは常に，ＢＩＵの命令キューから命令のオブジェクト・コードをフェッチして，バス・サイクルとは何の関係もないある数のクロック期間で命令を実行する．もし命令オブジェクト・コード・キューが空ならば，ＢＩＵは命令フェッチのマシン・サイクルを実行し，ＣＰＵは命令オブジェクト・コードがフェッチされるのを待つ．しかし，まもなく明らかになる理由から，キューはめったに空とはならない．したがって，ＥＵは，命令フェッチが行なわれる間，一般に待たなければならないことはない．

メモリまたはI/O素子に，命令の実行中にアクセスしなければならないならば，ＥＵはＢＩＵにその必要を通知する．ＢＩＵは，ＥＵの要求に応じて，適当な外部アクセスのマシン・サイクルを実行する．

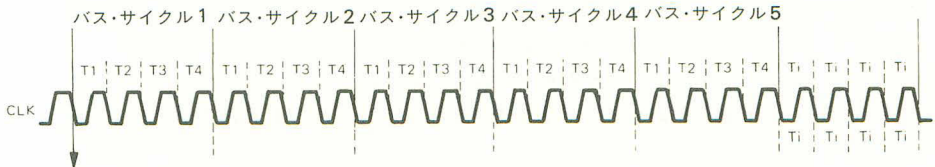
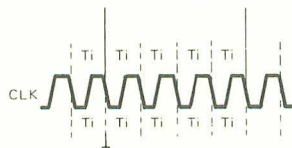
ＢＩＵはそれ自体，ＥＵと独立であり，6バイトのキューを命令オブジェクト・コードで満たしておくことを試みる．もしこの6バイトの2ないしそれ以上が空ならば，ＥＵにバス・アクセス保留の有効な要求がなければ，ＢＩＵは命令フェッチのマシン・サイクルを実行する．ＢＩＵが命令フェッチのマシン・サイクルの途中にある間に，ＥＵがバス・アクセスの要求を出すと，ＢＩＵはＥＵのバス・アクセスの要求を引き受ける前に，命令

フェッチのマシン・サイクルを完了する.

7.2.5 8086命令キュー

命令が実行されるときに何が起きるかについて考える. 簡単な場合から始めて, バス・インターフェイス内の命令オブジェクト・コード・キューは空とする. したがって, EUが命令を要求すると, BIUは命令の第1のバイトをフェッチするためのバス・サイクルを実行する.

この特定の命令は，2バイトのオブジェクト・コードを必要とすると仮定する．事情を簡単にして，次の命令バイトをフェッチするために直ちに実行されるもう1つのバス・サイクルを示す．

この命令は，メモリから1ワードのデータを読み込み，このデータを用いて算術演算操作を行なうと仮定する．命令は，アクセスすべきデータのメモリ位置の有効アドレスを計算するために，いくつかのクロックの期間を必要とする（7クロックの期間を要すると仮定する）．さらにまた，算術演算操作を行なうためにいくつかのクロックの期間が必要となる（9クロックの期間を仮定する）．通常のマイクロプロセッサでは，この命令は，次の一連のマシン・サイクルとして実行される．

しかし，非同期のＣＰＵとバス・コントロール・ユニットのロジックを持つ8086は，上に示した命令を実行するために，クロック期間を次のように用いる.

EU	EUはオブジェクト・コード・バイトを要求する．存在しないのでBIUは1バイトをフェッチする.	EUは第2のオブジェクト・コード・バイトを必要とする.	EUは7クロック期間でデータのメモリ・アドレスを計算する。7番目のクロックの終わりにCPUはバス・アクセスを要求する.	EUは要求したデータがBIUでフェッチされるのを待つ.
BIU	BIUは1バス・サイクルでオブジェクトの1バイトをフェッチする.	BIUは1バス・サイクルでオブジェクト・コードの第2のバイトをフェッチする.	EUはバス・アクセスを要求していないので，BIUは次の2つのオブジェクト・コード・バイトをフェッチしてキューにストアする．バス・サイクル4の終わりにEUはバス・アクセスを要求するので，BIUはCPUへのサービスを行なう.	BIUは，CPUによって示されたメモリ位置からデータをフェッチする.

EU	EUは必要な算術演算操作を実行するために9クロックの期間を用いる.	EUは，BIUのキューから次のオブジェクト・コード・バイトを取り出し，次の命令の実行を開始する.
BIU	BIUは，命令オブジェクト・コード・キューを満たすためにバス・サイクルの実行を続ける.	

ところで，記憶されているように，データ・アドレスが偶数バイト境界にあれば，8086は16ビットのインクリメントを行なってデータをフェッチするので，上に示した図は正確でない．また，ＢＩＵは，少なくともキューに２バイトの空があるときに限り，命令のバイトをフェッチしてキューにロードする．すべてのデータが偶数バイト境界にあると仮定すると，タイミングは次のようになる．

EU	EUはオブジェクト・コード・バイトを要求する．存在しないので，BIUは１バイトをフェッチする．	EUは7クロックの期間でデータのメモリ・アドレスを計算する．7番目のクロックの終わりにEUはバス・アクセスを要求する．	EUは要求したデータがBIUでフェッチされるのを待つ．	EUは算術演算操作を実行するために9クロックの期間を用いる．	
BIU	BIUは1バス・サイクルでオブジェクト・コードの2バイトをフェッチする．CPUはこの両方を取り出すので，キューはただちに空となる．	BIUは2バス・サイクルでオブジェクト・コードの4バイトをフェッチし，キューにストアする．キューには空の2バイトが残る．	BIUはEUによって示されたメモリ位置からデータをフェッチする．	BIUはさらにオブジェクト・コードの2バイトをフェッチしてキューにストアし，これでキューは満たされる．	BIUはアイドル状態となる．

EU	EUは命令の実行を終わり，次の命令を実行するためにキューから1バイトのオブジェクト・コードをフェッチする．
BIU	1バイトだけキューが空なのでBIUはアイドル状態のままである．

8086バス・サイクルのタイミングについて注意すべきいくつかの重要な点がある．バス・サイクルは，ＢＩＵの現象である．

ＥＵロジックに関する限り，バス・サイクルは存在しない．ＥＵは，命令を実行している活動期間と，命令のオブジェクト・コードあるいはＢＩＵがバス・サイクルによって処理するデータを待っている不活動期間を経験する．ＥＵの活動の期間は，一連のクロック期間で時間が定まる．ＥＵは，クロック期間をマシン・サイクルにまとめることを試みたりはしないし，クロック期間が特定の数値の組合せで生じることもない．

ＢＩＵに関する限り，データが8086との間で伝送されるべき場合にだけ，クロック期間はバス・サイクルにまとめられる．第１のプライオリティは，ＥＵから出されるバス・アクセスの要求に与えられている．ＥＵがバス・アクセスを要求しなければ，ＢＩＵはキューが満たされるまで，命令フェッチのバス・サイクルを実行する．以下は，ＢＩＵが命令フェッチのバス・サイクルを実行するために，前もって必要な事柄である．

1．バス・サイクルを初期化するクロック期間は，初期化の期間でなければアイドルのクロック期間である．

2．ＥＵは，保留の有効なバス・アクセス要求を持たない．

3．少なくともキューに２バイトの空きがある．

キューが満たされていれば，ＢＩＵはバス・サイクルの実行を中止し，前に示したように，一連のアイドル・クロック期間が生じる．

ＣＰＵはバス・アクセスを待たなければならないことに注意．前に示した図で，ＣＰＵは，データのメモリ・アドレスを計算するために，７クロックの期間を要する．７番目のクロック期間の終わりに，ＥＵはＢＩＵに対してバス・アクセスの要求を出す．しかしこの時点でＢＩＵは，命令フェッチのバス・サイクルの間，分離した状態にある．ＢＩＵは，命令フェッチのバス・サイクルを終了して，ＥＵのバス・アクセス要求を引き受ける．

前に示した最後の図で，新しい命令の実行開始にバス・サイクルは伴わない．実行される次の命令は１バイトのオブジェクト・コードを持つことを仮定している．このオブジェクト・コード・バイトはキューの前からフェッチされ，キューは，丁度１バイトの空きを持つ．キューからコードが取り出されるので，命令オブジェクト・コードをフェッチするためのバス・サイクルは実行されない．続いて，１バイトだけの空きが存在するので，ＢＩＵは命令フェッチのバス・サイクルを実行しない．ＢＩＵが命令フェッチのバス・サイクルを実行するには，キューに少なくとも，２バイトの空きが存在しなければならない．8086の命令フェッチのキュー処理についての前述の解説を基礎に，8086が本質的に命令フェッチの時間を省略していることが見られる．ＢＩＵが命令オブジェクト・コードをフェッチする間に，ＥＵがウエートしなければならないのは，条件付き分岐命令によってキューの系列からの分岐が実行されたとき，あるいは（何かの理由で）命令実行に伴うメモリ・アクセスがあまりにも頻繁で，命令フェッチのバス・サイクルを挿入するのに十分なアイドル・クロック期間がＢＩＵにないときに限る．

システム・デザインに影響する，キューに関するその他の情報については，レディの実現とタイミングの節の最後の所を参照のこと．

第8章
8086の基本デザイン

8.1　動作モード

8086は，様々で著しく異なるアプリケーションでも容易に動作するように，構成される．7章で述べたMN/$\overline{MX}$ 入力は，“ミニマム・モード”と“マキシマム・モード”として区別される出力の2つの異なる組として，8086を機能させる接続による選択となっている．

ここで2つのモードについてさらに詳しく検討する．

8.1.1　ミニマム・モード

ミニマム・モードの8086は，MN/$\overline{MX}$ ピンをV_{cc} に接続する．ミニマム・モードは，1または2枚のボードのシングルＣＰＵシステムで用いる．**図8-1**にミニマム・モードの8086を示す．

ミニマム・モードでは，8086は1メガバイトのメモリ領域と64キロバイトのI/O領域の全体をアドレス指定できる．データ・バスは16ビットの幅を持つ．8086は直接に，バス・コントロール(DT/$\overline{R}$, $\overline{DEN}$, ALE, M/$\overline{IO}$, $\overline{RD}$, $\overline{WR}$, $\overline{INTA}$）　を供給する．既存のＤＭＡコントローラと互換性のある，簡単なＣＰＵの権利獲得の機構は，HOLDとHLDAの信号によって可能となっている．

8.1.2　マキシマム・モード

図8-2に示されているマキシマム・モードは，MN/$\overline{MX}$ ピンをグランドに接続する．マキシマム・モードは，マルチプロセッサとコープロセッサ構成で用いられる．

マキシマム・モードにおいて，8288バス・コントローラは入力として8086からコントロール信号を受け取る．この入力は，コントロールの出力信号を生成するために，8288によってデコードされる．8086の他のコントロールの信号も，外部ロジックに情報を与えるた

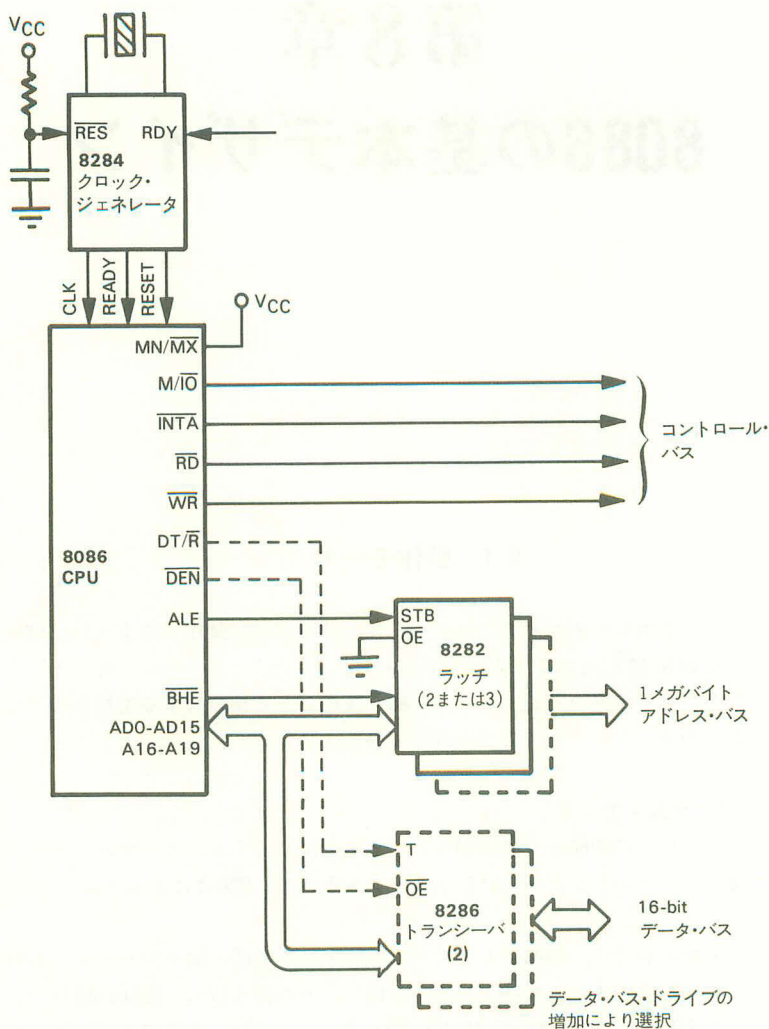



図8-1　ミニマム・モード8086

めに変更される．特に，8086の出力信号を次のように再定義する．

1．キュー・ステータスは，QS 0とQS1に出力される．これにより，外部素子，たとえばICE/86 あるいは特定の命令セット拡張のコープロセッサが，CPUの命令実行に追従することが可能となる．

2．システムのコントロールと構成のオプションは，バス・サイクル・ステータスの出力，$\overline{S0}$，$\overline{S1}$，$\overline{S2}$ によって拡張される．これらの出力は，8288バス・コントローラ，

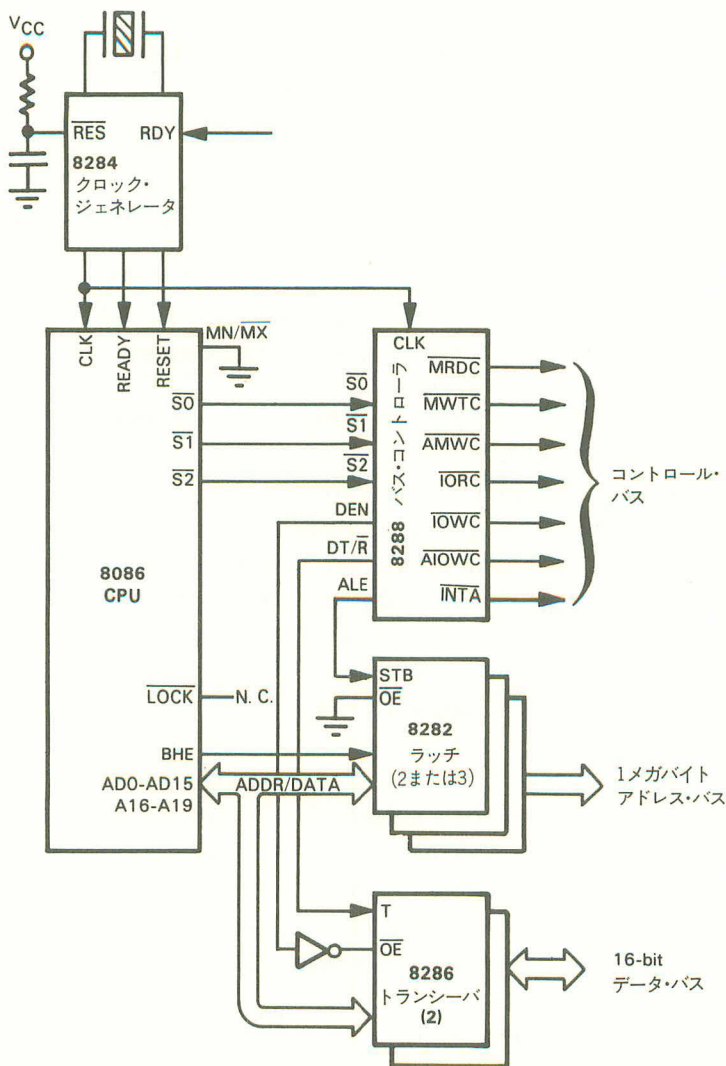



図8-2　マキシマム・モード8086

8289バス・アービタ* そして類似の外部素子によって用いられる．

3．マルチプロセッサ・システムにおいて共有資源に対するアクセスの制御は，バス・ロックの機構によってサポートされている．

4．プロセッサ獲得の２つの優先順位のあるレベル（$\overline{RQ}/\overline{GT0}$，$\overline{RQ}/\overline{GT1}$）によって，システム・バス・インターフェイスを共有する8086のローカル・バス上に，複数プロ

＊　arbiter：(権利)判定用素子（訳者注）．

セッサの存在が可能となる．

次に，これらの拡張された能力がどのように用いられるかを検討する．

キュー・ステータスは，内部のキューからどのような情報が移動されているか，コントロールが移されたことによって，いつキューがリセットされるかを示す．**表8.1**に，キュー・ステータスの意味の要約を示す．

表8-1　キュー・ステータスの出力

QS1	QS0	意　　　味
0	0	ノー・オペレーション
0	1	キューからのオペコードの第1のバイト
1	0	キューが空
1	1	キューからの後続のバイト

キューの操作が行なわれた後のCLKサイクルの間キュー・ステータスは有効.

図8-3に示されているものと同様のロジックを用いれば，8086のキュー・ステータスを追跡することができる．$\overline{S0}$，$\overline{S1}$，$\overline{S2}$がそれぞれ1, 0, 0のとき，命令フェッチが実行されている．QS0とQS1は，命令が8086のキューあるいは外部メモリのどちらからフェッチされているかを示す．外部メモリのアクセスに対して，A0と$\overline{BHE}$はワードあるいはバイトのアクセスを表わす．このロジックは多くの方法で用いることができる．以下の例を考える．

ICE/86は，特定のメモリ位置にストアされた命令の実行を追跡できる．キューを追跡するためにICE/86によって用いられる回路の例を図8-3に示す．

第1のアップダウン・カウンタはキューの深さを追跡し，第2のアップダウン・カウンタは釣り合うキューの深さを補える．第2のカウンタは，キューが空になるまで，キューからさらにフェッチするに従って減少するか，釣り合うアドレスの実行を表示して，カウントは0になる．第1のカウンタは，キューからのフェッチ（QS0＝1）によって減少し，キューにオブジェクト・コード・バイトがストアされると増加する．外部メモリからの通常の命令フェッチはキューに2バイトを移動し，カウンタは2つのクロックの増加（T201とT301）となることに注意．バスの上位（A0がハイで$\overline{BHE}$がロー）から1バイトがロードされたときは，カウンタは1回増加する．ＥＵはＢＩＵに同期していないので，キューからのフェッチはキューへの移動と同時にも起きる．第1のカウンタの$\overline{ENP}$入力を駆動するエクスクルーシブＯＲゲートにより，同時操作を互いにキャンセルしてキューの深さを変更しないようにしている．

8086のキューは，ESCAPE命令の実行を検出するためにコープロセッサによって追跡される．ESCAPE命令は，コープロセッサにある特殊なタスクの実行を指令する．

表8-2　ステータス・ラインの出力

$\overline{S2}$	$\overline{S1}$	$\overline{S0}$	意　　　味
0	0	0	インタラプト・アクノリッジ
0	0	1	リードI/Oポート
0	1	0	ライトI/Oポート
0	1	1	ホルト
1	0	0	コード・アクセス
1	0	1	リード・メモリ
1	1	0	ライト・メモリ
1	1	1	パッシブ

表8-2に，ステータス・ライン$\overline{S0}$，$\overline{S1}$，$\overline{S2}$の意味を定義する．前に述べたように，これらのステータス・ラインは，いつバス・サイクルを初期化するか，どのようなタイプのコマンドを発行するか，そしていつバス・サイクルを終了するかを，8288に通知する．8288は，ＣＰＵクロック期間の開始に

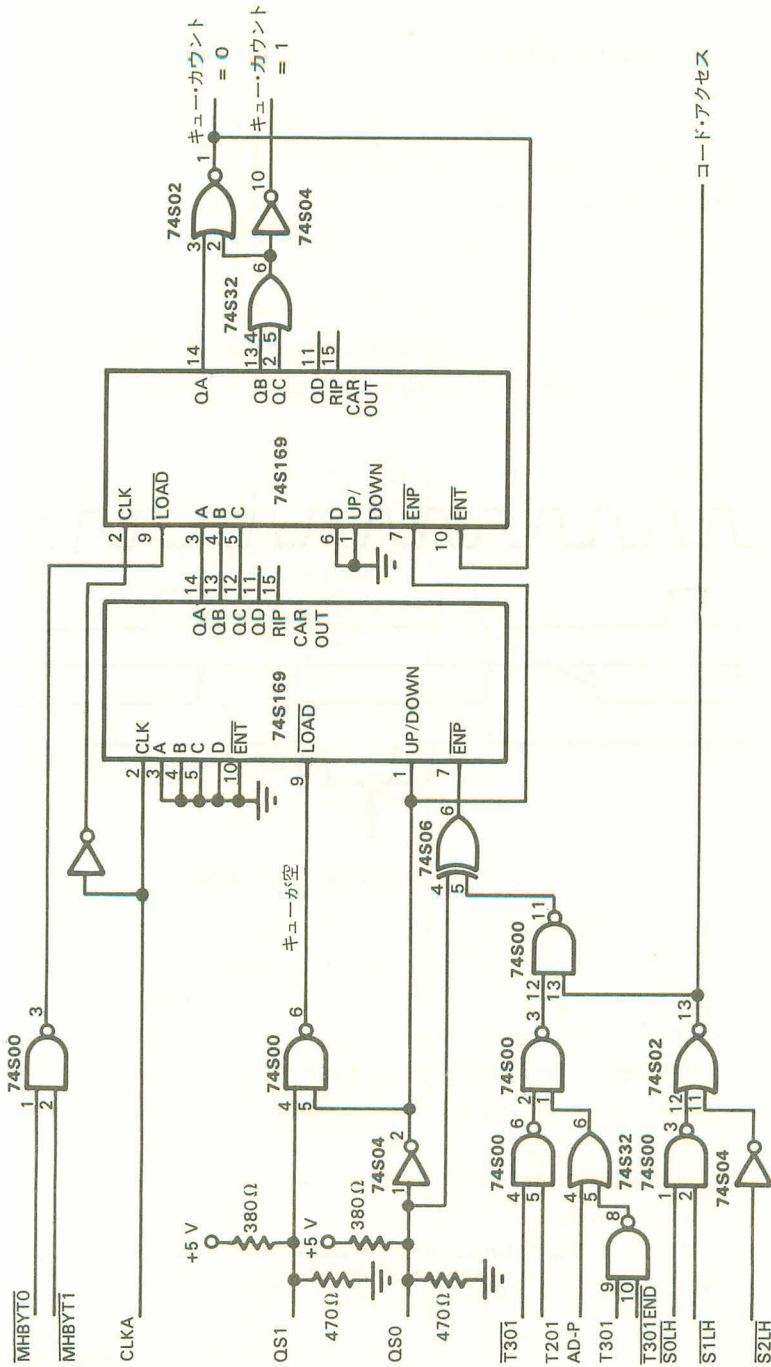



図8-3　8086キュー追跡の回路

ステータス・ラインをサンプルする．バス・サイクルの初めに，ＣＰＵはステータス・ラインをパッシブ状態($\overline{S0}$，$\overline{S1}$，$\overline{S2}$がすべてハイ）から可能な７つのアクティブ状態の１つに変える．

新しいバス・サイクルに対して，以前のバス・サイクル期間Ｔ４のクロックの立ち上がりのエッジ，または現在バスが動作していなければTI のアイドル・サイクルの間に，8086は$\overline{S0}$，$\overline{S1}$，$\overline{S2}$の状態を変える．8288は，クロックのハイからローへの遷移でステータス・ラインをサンプルすることによって，ステータスの変化を検出する．

8288は，適当なバッファ方向のコントロールを伴い，ハイのＡＬＥパルスを生成することによってバス・サイクルを開始する．これは，ステータス変更の検出直後のクロック期間に生じる．バス・トランシーバと指定された操作は，次のクロックの期間，動作可能となる．ステータスがパッシブ状態に戻ると，8288は動作を終了する．タイミングを **図8-4** に示す．

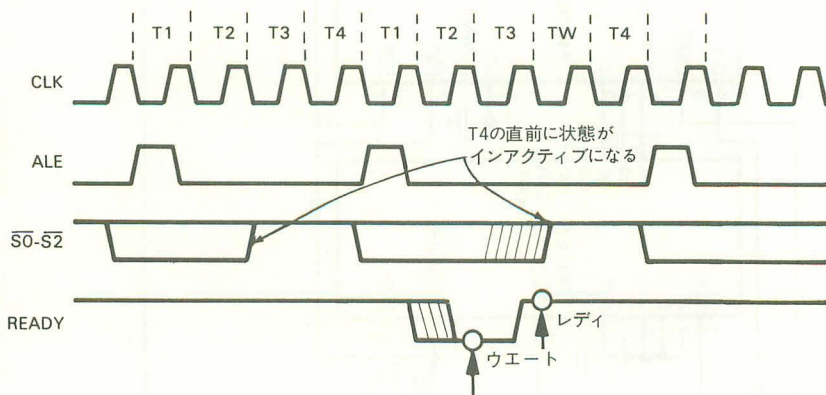



図8-4　ステータス・ラインのアクティブ化と終結

8086は，ウエート状態の間，$\overline{S0}$，$\overline{S1}$，$\overline{S2}$のレベルを維持する．前に述べたように，8284クロック・ジェネレータにローのＲＤＹ信号を入力する外部ロジックによって，ウエート状態が引き起こされる．クロック・ジェネレータはハイのREADY信号を出力するが，これはCLKとの同期がとられて8086に伝達される．

8086はウエート状態の間，$\overline{S0}$，$\overline{S1}$，$\overline{S2}$のレベルを維持するので，任意のクロック期間にわたるウエート状態に対して，8288はアクティブなバス・コントロールを保持する．バスの動作をモニタし，他のプロセッサがローカル・バスのコントロールを獲得したならば8288をコントロールするために，8086のローカル・バス上の他のプロセッサによっても，ステータス・ラインは用いられる．

8288は，バス・コントロール信号DEN，DT/$\overline{R}$，ALEとコントロール信号$\overline{INTA}$，$\overline{MRDC}$，$\overline{IORC}$，$\overline{MWTC}$，$\overline{IOWC}$，$\overline{AIOWC}$を供給する．コントロール信号は，Intel MULTI-BUSとの互換性のために，メモリとI/Oに対するリード/ライトの操作を分離する．

周辺装置やスタティックＲＡＭによってしばしば要求される広いライト・パルス幅を供

給するために，通常のライト・コントロール信号より1クロック期間速い，アドバンスド・ライト・コントロール信号が使用できる．通常のライト・コントロール信号により，ライト・コントロール・パルスの立上りエッジでデータのストローブを行なうダイナミックRAMメモリとI/O素子に適応するために，ライト・パルスの前にデータが設定される．アドバンスド・ライト・コントロール信号は，コントロール信号の立上りエッジより前で，データが有効であることは保証していない．

マキシマム・モードにおけるDEN信号は，ミニマム・モードと比較すると反転している．これはDENを，他の信号，特にインタラプト・コントロールとロジック・ゲートで組み合わせることを容易にしている．**図8-5**に，ミニマム・モードとマキシマム・モードのバス転送コントロール信号のタイミングの比較を示す．

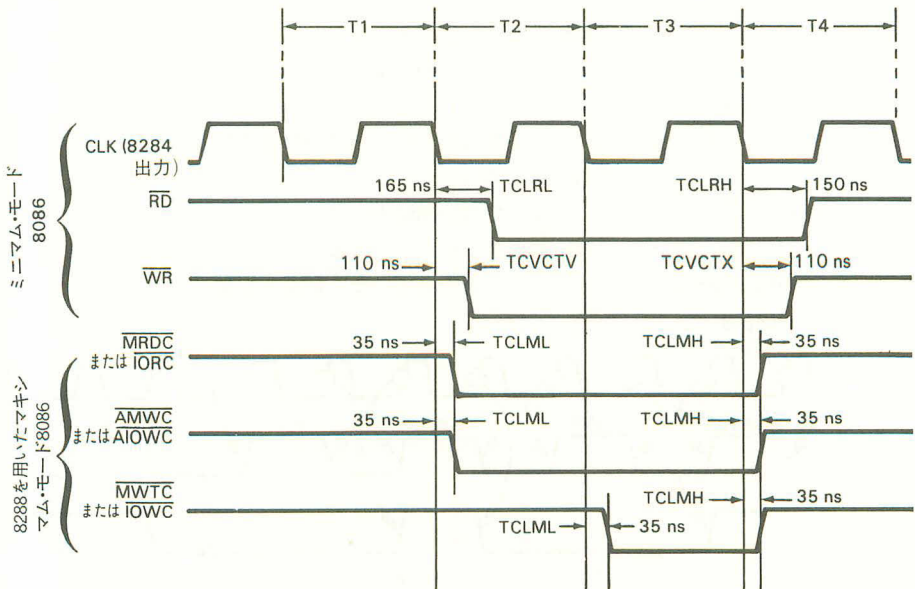



図8-5　ミニマムとマキシマムのバス転送タイミング

マキシマム・モードは，多重処理構造と，規模の大きいシングルCPUデザイン（マルチバス・システムあるいは，2つまたはそれ以上のPCボードを含むシステム）のために設計されている．8288はバイポーラ素子である．したがって，コントロール信号の32 mAの駆動能力と，タイミング・パラメータの許容誤差と最悪の場合の遅延から，ミニマム・モードの8086に比較して規模の大きい良い性能のシステムが得られる．

8086から取り除かれた機能に加えて，8288は，マルチプロセッサ構成と8086のローカル・バスの周辺素子をサポートするための付加的な接続による選択とコントロールを与える．この能力は，メモリまたはI/Oを含む資源を共有あるいはローカルなものとして割り当てることを可能にする．共有の資源は，Multibus システム・バスで利用できる．ローカルな

資源は，そのローカル・バス上の8086によってだけアクセスできる．この技法によって，Multibusシステム・バスに対するアクセスの競合が減少し，マルチCPUシステムの性能が改善される．特殊な構成については後の章で述べる．

8086マキシマム・モードの$\overline{\text{LOCK}}$出力は，共有資源に対するアクセスのコントロールに役立つ．$\overline{\text{LOCK}}$の出力は，8086がLOCKプレフィックス命令を実行すると，アクティブになる．$\overline{\text{LOCK}}$の出力は，LOCKプレフィックスの実行に続く最初のクロックの期間でローになり，LOCKプレフィックスに続く命令の最後の命令実行クロック期間中と次の命令実行の最初のクロック期間でローになっている．$\overline{\text{LOCK}}$の信号は，すべてのマイクロプロセッサのシステム・バス判定ロジックの一部でなければならない．

通常のマルチプロセッサ・システムの動作の間，共有のシステム・バス・アクセスの優先権は，1サイクルごとを基礎に判定回路によって決定される．ある8086にシステム・バスを通してデータを伝送する必要が生じると，バス・アクセスを要求する．特定のシステム・バス判定方法によって決定されて，この8086が優先権を獲得すると，システム・バスのコントロールを得てバス・サイクルを実行し，次に，システム・バスのコントロールを保持するか，自主的にシステム・バスを解放するか，あるいは優先権を失なうことによって強制的にシステム・バスから切り離される．

LOCK機構は，自主的であってもなくても，8086がシステム・バスのコントロールを失なうことを防止する．これにより，他のCPUによる干渉と可能性のあるデータの破壊を受けずに，8086が複数のバス・サイクルの命令を実行する能力を保証している．LOCKの出力の動作を図8-6に示す．

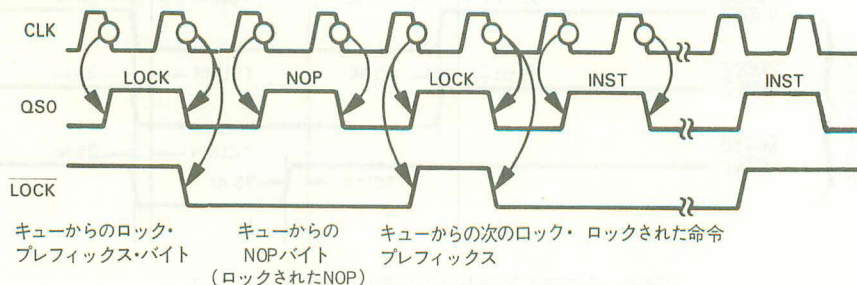



図8-6　LOCKの動作

$\overline{\text{LOCK}}$の出力は，離れてロックされた命令の間ではインアクティブになることに注意．また，LOCKプレフィックスは実行時間に2つのクロック期間を付加している．

キュー・ステータスは以前のクロック期間のキューの動作を反映しているので，$\overline{\text{LOCK}}$の出力は，実際に次の（ロックされた）命令の開始に一致してアクティブとなり，ロックされた命令の実行に続く1クロックの間，アクティブとなる．

LOCKプレフィックスに続く命令のオブジェクト・コードがキューになければ，示したように命令のオブジェクト・コードが外部メモリからフェッチされる間，$\overline{\text{LOCK}}$の出力はローとなる．

バス・インターフェイス・ユニット(BIU)は，ロックされた命令の実行期間も命令フェッチ・サイクルを実行する．LOCKは単に，命令実行の間，1つの8086がシステム・バスのコントロールを保持することを保証するだけで，このCPUがこのロックの期間に行なうことの可能なバス動作のタイプを決して制限するものではない．

LOCK機構は，TESTとSETのハンドシェイク・シーケンスで一般に用いられる．このシーケンスの間，8086は共有メモリ位置から読み出しを行ない，その位置にデータを戻す．他のどのCPUも，リード操作のTESTとライト操作のSETの間は，このメモリ位置を参照することができない．8086はこの動作を次のようにロックされた交換命令で行なう．

```
LOCK   XCHG   reg, memory   ; reg は8086の任意の
                            ; レジスタ, memory は信号
                            ; のアドレス
```

マルチプロセッサ・システムにおいて，LOCKの興味ある別の用途は，1つのCPUのメッセージ・バッファから他へ高速のメッセージ転送を可能とする，ロックされたブロックの移動である．

ロックされた命令の期間，$\overline{RQ}/\overline{GT}$ラインによって発生するプロセッサ獲得の要求は記録されるが，ロックされた命令の完了までは受け付けられない．

LOCKプレフィックスは，割り込みに対して直接的影響を持たない．一般には，プレフィックスのバイトは，それが先行する命令の拡張と考えられる．したがって，プレフィックスの実行中に発生した割り込みは，プレフィックスに続く命令の完了まで受け付けられない（割り込みは有効であると仮定）．

複数のプレフィックスのバイトが1つの命令に先行できることに注意．繰返しのプレフィックス(REP)は，続く命令の実行の度に割り込みで中断できる．このことは，REPがLOCKプレフィックスと結合されても成り立つ．したがって，割り込みはブロック移動あるいは繰返しのストリング操作で，ロックの影響を受けない．この動作と複数プレフィックスのストリング操作についてのこれ以上の情報は，この章の後の8086インタラプト構造を取り扱う節で述べる．

優先づけられたプロセッサ獲得の付加的レベルについては，この章の後で詳細に述べる．

8.2 クロックの発生

8086は，ローが−0.5から＋0.6までとハイが＋3.9からV_{cc}＋1.0の電圧の間の，速い立上りと立下りの時間（最大10ナノ秒）を有するクロック信号を必要とする．8086の最大クロック周波数は5MHzである．8086の設計はダイナミック・セルを取り入れているので，2MHzの最低周波数が必要とされる．最低周波数条件から，CPUのシングル・ステップあるいはサイクリングは，クロックを無効にすることによっては果たされない．CPUクロックのタイミングと電圧の必要条件を図8-7に示す．

一般に，最大値以下の周波数に対して，CPUのクロックは，8086のデータ・シートに述べられている周波数依存のパルス幅制限を満たす必要はない．指定されている値は満た

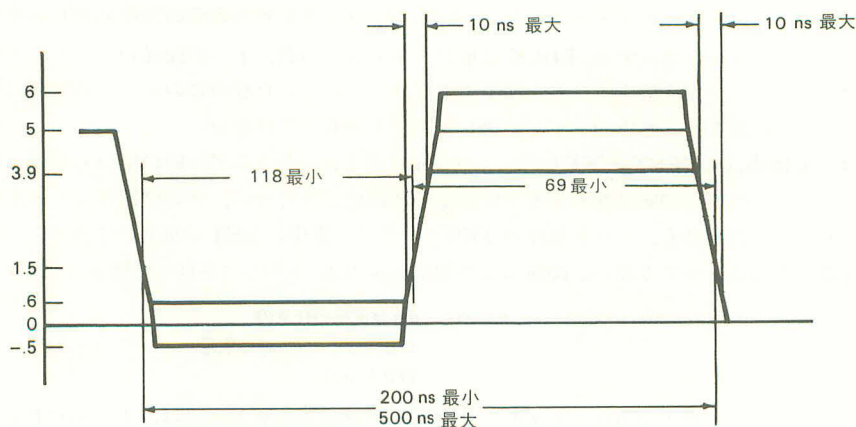



図8-7 8086 CPUのタイミングと電圧の必要条件

されるべき最小値を反映しているだけで，最大クロック周波数の立場から述べられている．クロック周波数がＣＰＵの最大周波数に近づくと，ＣＰＵの最小クロックのローとハイの時間の仕様を満たすために，クロックは33%のデューティ・サイクルに適合しなければならない．

必要な電圧レベルと遷移時間を有する最適な33%デューティ・サイクルのクロックは，**図8-8**に示すように，8284クロック・ジェネレータで得ることができる．

外部周波数源あるいは直列共振クリスタルで8284が駆動される．選ばれたソースは，必要なＣＰＵ周波数の３倍（３×）で発振しなければならない．クロック発生の周波数源として8284のクリスタル入力を選ぶためには，8284への $F/\overline{C}$ 入力はグランドに接続されなければならない．この接続による選択で，クロック・ジェネレータのソースとして，クリスタルまたは外部周波数の入力を選ぶことができる．8284はオーバトーン・モードのクリスタルに適応するタンク回路の入力を有しているが，より正確な（そして安定な）周波数発生のために基本モードのクリスタルを推奨する．8284に用いるためにクリスタルを選ぶときは，直列抵抗はできるだけ低いことが必要である．

他の回路要素は動作周波数を共振から変化させる傾向があるので，動作インピーダンスは指定された直列抵抗よりも一般に高くなっている．オシレータの帰還回路の減衰がループ利得を１以下に下げると，オシレータは動作しなくなる．

電圧と温度の変動に対してオシレータの安定性を最も良くするためには，Ｘ１とＸ２へのクリスタルの接続に510Ωの抵抗で接地することを勧める．インテル

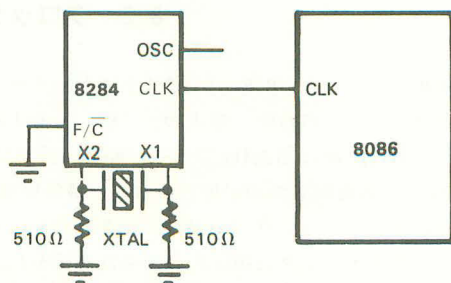



図8-8 8284を用いたCLKの供給

のマイクロプロセッサのクリスタルを供給している多くのベンダーの2社を**表8-3**に，関連のあるいくつかの周波数のクリスタル部品番号と共に示す．

表8-3　クリスタル・ベンダー

f	パラレル/シリーズ	Crystek[1] Corp.	CTS Knight,[2] Inc.
3.6 MHz	P	**	**
5.185 MHz	S	CY8A	**
6.0 MHz	P	**	MP060
6.144 MHz	P	**	MP061
6.25MHz	P	**	MP062
10.0 MHz	P	**	MP10A
15.0 MHz	S	CY15A	MP150
18.432 MHz	S	CY19B*	MP184*
24.0 MHz	S	**	MP240
25.0 MHz	S	**	MP250
27.0 MHz	S (オーバトーン)	CY27A	MP270

*　このアプリケーションに対してはIntelもクリスタル番号8801を供給している．
**　適当な仕様でベンダーに連絡のこと．
注)　1.住所：1000 Crystal Drive, Fort Meyers, Florida 33901
　　2.住所：400 Reimann Ave., Sandwich, Illinois

高精度の周波数源，外部可変の周波数源，あるいは複数の8284を駆動する共通のソースが必要ならば，**図8-9**に示すように，1キロオームを通してF/$\overline{C}$入力を＋5ボルトに接続することによって，8284のエクスターナル・フリーケンシィ・インプット（ＥＦＩ）を選ぶことができる．

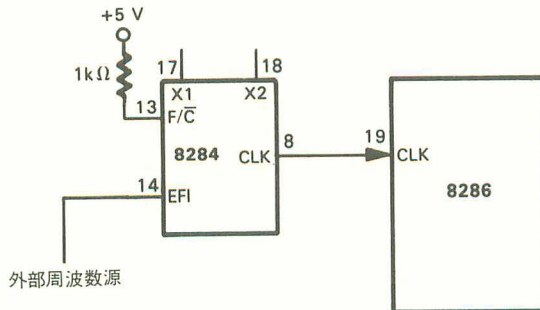



図8-9　外部周波数源の利用

外部周波数源は，ＴＴＬと互換性があり，50%のデューティ・サイクルを持ち，必要なＣＰＵ動作周波数の3倍で発振しなければならない．8284が受け付けられる最大ＥＦＩ周波数は，クロックのローとハイの時間が最小13nsで，24MHzよりも少し高い．最小ＥＦＩ周波数は指定されていないが，ＣＰＵの最小クロック速度に違反してはならない．

システムに分散する8284を駆動するために共通の周波数を用いるならば，各8284はソースからの個別のラインで駆動されなければならない．システムのノイズを最小にするため，

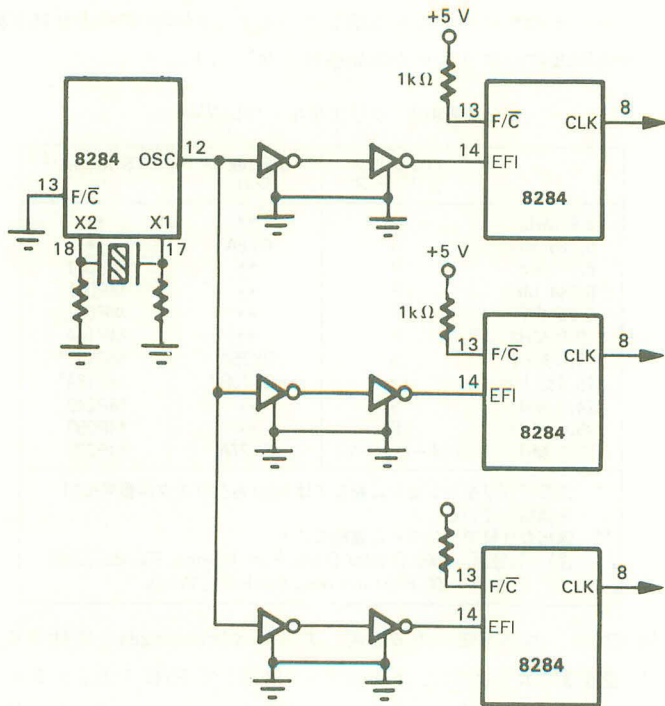



図8-10　マスタ周波数源の発生

各ラインは，74LS04 などのバッファで駆動されるツイスト・ペアとし，そのグランドはソースとレシーバのグランドに接続しなければならない．クロックのスキューを最小にするためには，すべての8284に対するラインの長さは等しくなければならない．付加された8284に対するメインの周波数源を発生する簡単な方法を図8-10に示す．

図8-10において，1つの8284はクリスタルで必要な周波数を発生させるために用いられている．8284のオシレータ出力（ＯＳＣ）はクリスタルの周波数に等しく，システムの他のすべての外部周波数入力を駆動するために用いられる．

オシレータの信号のコンプリメントがＣＰＵクロック・ジェネレータ回路の駆動に用いられるように，ＯＳＣは反転している．したがって，２つの8284が同期すべき別々のＣＰＵのクロック入力を駆動するならば，１つの8284のＯＳＣでもう１つの8284のＥＦＩ入力を駆動することはできない．8284の範囲でＥＦＩ対ＣＬＫの遅延の変動は 35ns から 45ns に近づく，しかし，すべての8284が同一のパッケージ・タイプで，同一の相対供給電圧で，同じ温度環境で動作するならば，変動は 15ns と 25ns の間にまで減少する．

8284には，前述のＯＳＣ，ＣＰＵを駆動するシステム・クロック（ＣＬＫ），そしてＣＰＵのクロック周波数の½で動作する周辺装置用クロック（PCLK）の３つの周波数出力がある．ＯＳＣはクリスタルで駆動されているだけで，F/$\overline{C}$ の接続による選択の影響を受けな

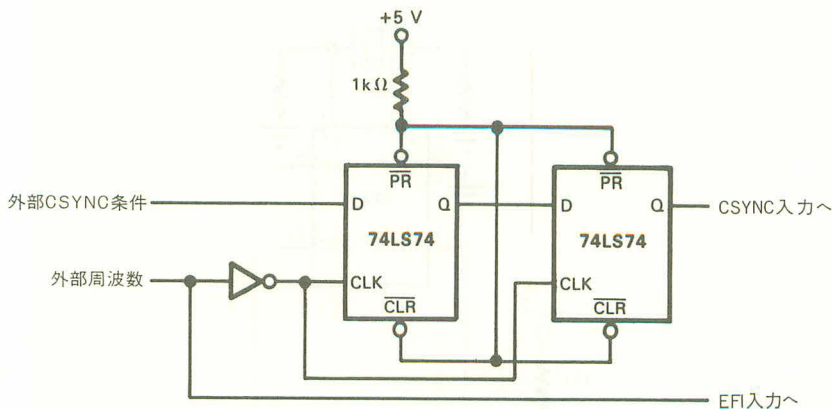



図8-11　CSYNCの同期

い．外部周波数入力が用いられてクリスタルが8284に接続されていなければ，ＯＳＣは不定となる．ＣＬＫは，選択された周波数源から３進カウンタの内部分周によって得られる．カウンタは，最大周波数のＣＰＵに最適な33％デューティ・サイクルのクロックを発生する．ＰＣＬＫは50％デューティ・サイクルで，ＣＬＫの周波数の½で動作する．

３進カウンタで分周される8284の状態はシステム初期化（パワー・オン）では不定なので，ＣＰＵクロックの外部事象に対する同期が可能となるように，カウンタに対する外部同期信号（CSYNC）が備わっている．CSYNC がハイとなると，ＣＬＫとＰＣＬＫの出力は強制的にハイとなる．CSYNC がローに戻ると，同波数源の次の正のクロックでクロックの発生が開始する．CSYNC は，周波数源の最小２つの期間，アクティブでなければならない．CSYNC が周波数源と同期していなければ，同期のために**図8-11**の回路を用いる必要がある．

２つのラッチは，CSYNC を駆動するラッチにおいて準安定状態の可能性を最小にしている．**図8-12**に示すようにラッチは，周波数源に対する8284のセット・アップとホールド

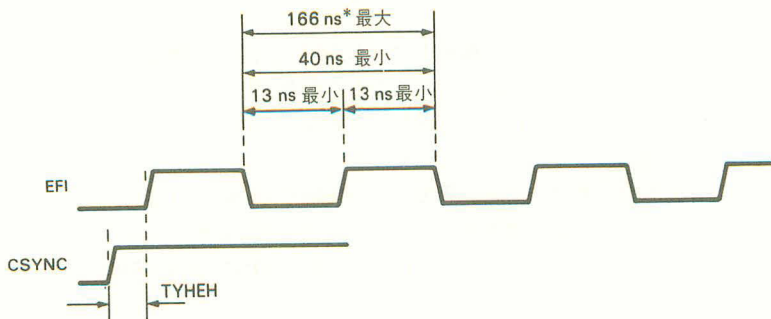



図8-12　CSYNCのタイミング

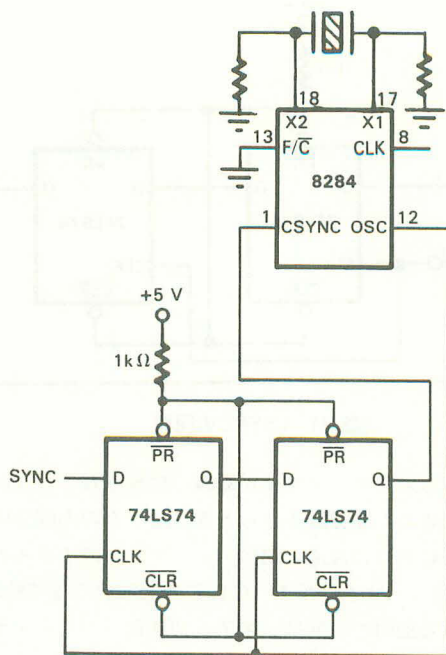



図8-13　OSCを用いたCSYNCの同期

の時間を保証するために，周波数源の反転をクロックとしている．

1つの8284が外部事象に同期の必要があり，外部周波数源が用いられていなければ，**図8-13**に示すように，8284のＯＳＣがCSYNCを同期させるために用いられる．

ＯＳＣは内部オシレータの信号に関して反転しているので，前の例でのインバータは必要でない．複数の8284を同期させる必要があれば，**図8-14**に示すように，外部周波数ですべての8284を駆動し，1個のCSYNC同期回路ですべての8284のCSYNC入力を駆動しなければならない．

CSYNCがアクティブのとき，8086クロックのローの最小時間は十分でないので，リセットの期間だけあるいはＣＰＵのクロックがハイの間は，イネーブルとしなければならない．CSYNCはまた，適当なＣＰＵのリセットを保証するために，リセットの終了前に最小4クロックの期間，ディスエーブルでなければならない．

8284のＣＬＫ出力における高速の変化と高い駆動力（5 mA）のために，リンギングの除去にクロック・ラインと直列に100オームの抵抗を入れる必要がある．スキューが最小のＣＬＫの複数のソースが必要ならば，ＣＬＫは，最小VCH＝3.9（8086の入力がハイの最小電圧）を保証するために，出力が100オームを通して5ボルトに接続された，高い駆動力の素子（74S241）によるバッファを用いることができる．8284は，ＣＰＵに対するREADYの同期をとり，1つのＣＰＵに対するREADYを供給するだけなので，1つの8284は複数のＣＰＵのＣＬＫを発生するためには用いることができない．

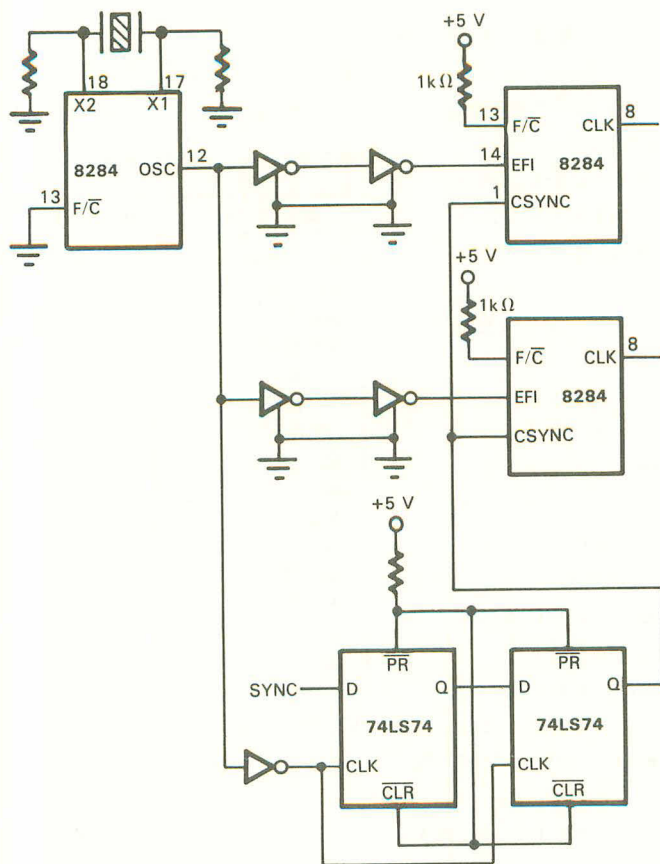



図8-14　複数の8284に対するCSYNCの分配

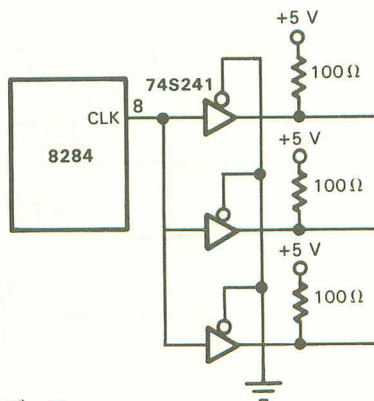



図8-15　高駆動力素子によるCLKのバッファ

8.3 リセット

8086は，50μsのリセット・パルスを必要とするパワー・オン後を除いて，最小パルス幅が４つのＣＰＵクロック期間のアクティブ・ハイのリセットを必要とする．ＣＰＵは内部でリセットとクロックの同期をとるので，外部リセットがローになった後の１クロックの期間まで，リセットは内部的にアクティブである．ノンマスカブル・インタラプト（ＮＭＩ），ミニマム・モードのホールド・リクエスト，あるいは内部リセットの間に生じるマキシマム・モードのＲＱパルスは，受け付けられない．内部リセット直後にアクティブなミニマム・モードのホールド・リクエスト，あるいはマキシマム・モードのＲＱパルスは，最初の命令フェッチの前に受け付けられる．

表8-4　リセット期間の8086バス信号

信　　号	状　　態
AD0-AD15	トライステート
A16-A19/S3-S6	不　定
$\overline{\text{BHE}}$/S7	不　定
S2/(M/$\overline{\text{IO}}$)	"1"にした後にトライステート
S1/(DT/$\overline{\text{R}}$)	"1"にした後にトライステート
S0/($\overline{\text{DEN}}$)	"1"にした後にトライステート
$\overline{\text{LOCK}}$/$\overline{\text{WR}}$	"1"にした後にトライステート
$\overline{\text{RD}}$	"1"にした後にトライステート
$\overline{\text{INTA}}$	"1"にした後にトライステート
ALE	0
HLDA	0
$\overline{\text{RQ}}$/GT0	1
$\overline{\text{RQ}}$/GT1	1
QS0	0
QS1	0

8086がリセットを認めると，ＣＰＵはバスを表8-4に示すような状態にする．

リセットが検出されると，ＣＰＵによって，多重化バス信号の接続はフロート状態となる．フロート状態にできる他の信号は，トライステートになる前のＣＬＫの１つのロー状態の間，インアクティブ状態に駆動される．この様子を図8-16に示す．

ミニマム・モードにおいて，ＡＬＥとＨＬＤＡはインアクティブに駆動されるがフロートとはならない．マキシマム・モードでは，RQ/GT ラインはインアクティブに保持され，キュー・ステータス出力（Q0 と Q1）はアクティブでないことを示す．キュー・ステータスはキュー・リセットを表示しないので，キューをモニタするユーザ定義の外部回路もまた，

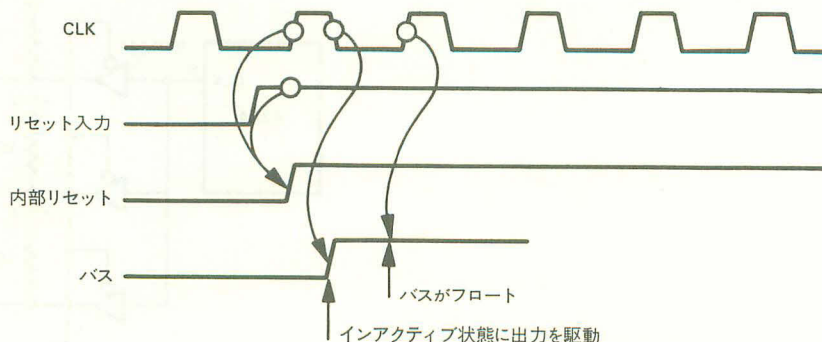



図8-16　リセットにおける8086バスの状態

システム・リセットによってリセットされなければならない．22キロオームのプルアップ抵抗をＣＰＵのコマンドとバス・コントロールのラインに接続する必要がある．これにより，漏れ電流（あるいはバス・キャパシタンス）がシステムにおける素子の最小のハイ電圧以下に電圧レベルを下げてしまうシステムで，上記ラインのインアクティブ状態が保証される．

マキシマム・モード・システムでは，8288は$\overline{S0}$－$\overline{S2}$入力に内部プルアップを含む．これにより，ＣＰＵがバスをフロート状態にしたときに，これらのラインのインアクティブ状態を維持する．リセットの間のステータス・ラインのハイ状態によって，8288はリセットのシーケンスをパッシブ状態として取り扱う．パッシブ状態に対する8288の出力状態を**表8-5**に示す．

表8-5　パッシブ状態の8288の出力

ALE	0
DEN	0
DT/$\overline{R}$	1
MCE/$\overline{PDEN}$	0/1
コマンド	1

バス・サイクルの間にリセットが起きると，ステータス・ラインはパッシブ状態に戻り，バス・サイクルは終わり，そしてコマンド・ラインはインアクティブになる．8288は，ステータス・ラインのパッシブ状態に基づいてコマンドの出力をフロートにしないことに注意．シングルＣＰＵシステムにおけるリセットの間に，ＣＰＵをバスから切り離す必要があれば，**図8-17**に示すように，リセット信号はまた8288のＡＥＮ入力とアドレス・ラッチの出力イネーブルに接続しなければならない．

この方法は，8288からのＤＥＮのインアクティブ状態がデータ・バスのトランシーバをフロートにする間，コマンドとアドレスのバス・インターフェイスをフロートする．

マイクロプロセッサ共有バス結合を確立するために判定を用いる複数プロセッサ・シス

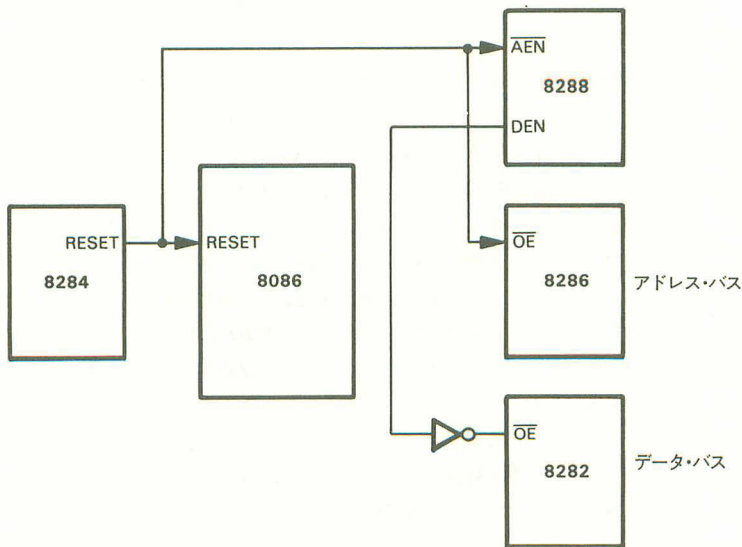



図8-17　マキシマム・モード8086バス・インターフェイスのリセット・ディスエーブル

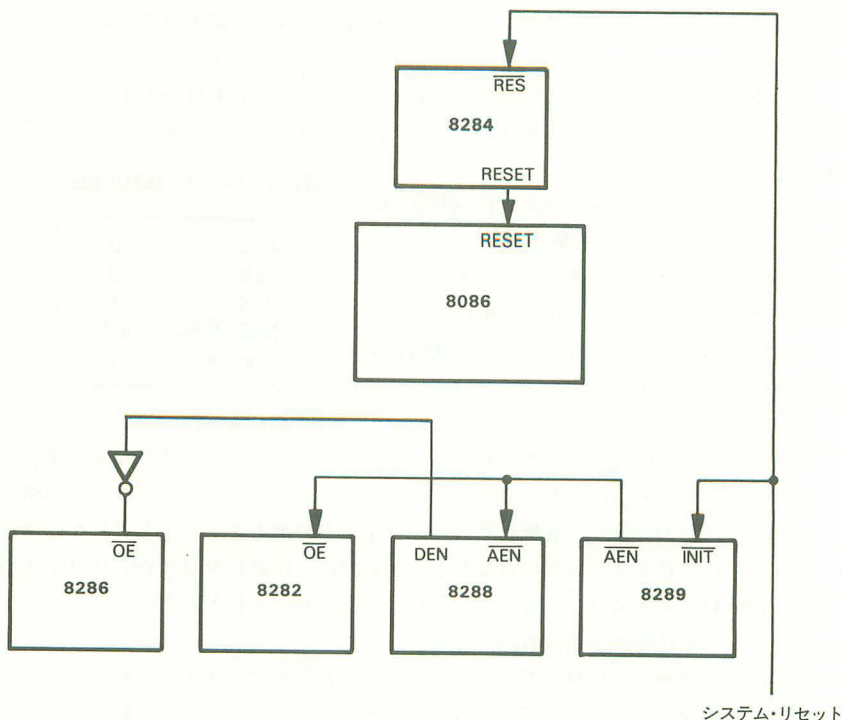



図8-18　マルチCPUシステムにおけるマキシマム・モード8086バス・インターフェイスのリセット・ディスエーブル

テムにおいて，**図8-18**に示すように，システム・リセットは，8284のリセット入力に加えて，8289バス・アービタのＩＮＩＴ入力に接続されなければならない．

アクティブ・ローのＩＮＩＴ入力により，すべての8289の出力はインアクティブ状態となる．インアクティブ状態の8289ＡＥＮ出力は，8288のコマンド出力をフロート状態にさせ，さらに，アドレス・ラッチはアドレス・バス・インターフェイスをフロートにする．１つ以上のマイクロプロセッサがマスタとして機能するマルチマイクロプロセッサ・システムでは，リセットはすべてのＣＰＵ，8289，8284に共通でなければならず，このリセットはすべてのＣＰＵのリセット条件，あるいは３つの8289バス・クロック期間（TBLBL）と３つの8086クロック期間を満たす必要がある．これは8289のリセット条件を満たしている．

リセットの間は8288のコマンド出力はフロートとなり，コマンド・ラインは2.2キロオームの抵抗を通して V_{CC} に接続されていなければならない．

8086に対するリセット信号は8284から得られる．8284は，アクティブ・ローの外部リセットからリセットを生成するために，8284はシュミット・トリガ入力を有している．8284データ・シートに記されているヒステリシスは，少なくとも0.25ボルトが8284リセット入力の０と１のスイッチング・ポイントを分離することを示している．ヒステリシスのない

入力は，近似的に同一電圧のスレッショルドでローからハイとハイからローにスイッチする．

入力は，明記されたローとハイの電圧（VIL と VIH）でスイッチすることが保証されているが，実際のスイッチング・ポイントはこの間のどこかにある．VIL_{min} は0.8ボルトと指定されているので，ヒステリシスにより，入力が少なくとも1.05ボルトに達するまで，リセットがアクティブであることが保証される．リセット入力である2.6ボルトのＶＩＨ以下に少なくとも0.25ボルト，入力が下がるまでリセットは認められない．

パワーアップからのリセットを保証するためには，リセット入力は，V_{cc} が4.5ボルトの最小供給電圧に達した後に，50マイクロ秒の間は1.05ボルト以下でなければならない．ヒステリシスにより，リセット入力は図8-19に示す簡単なＲＣ回路で駆動できる．

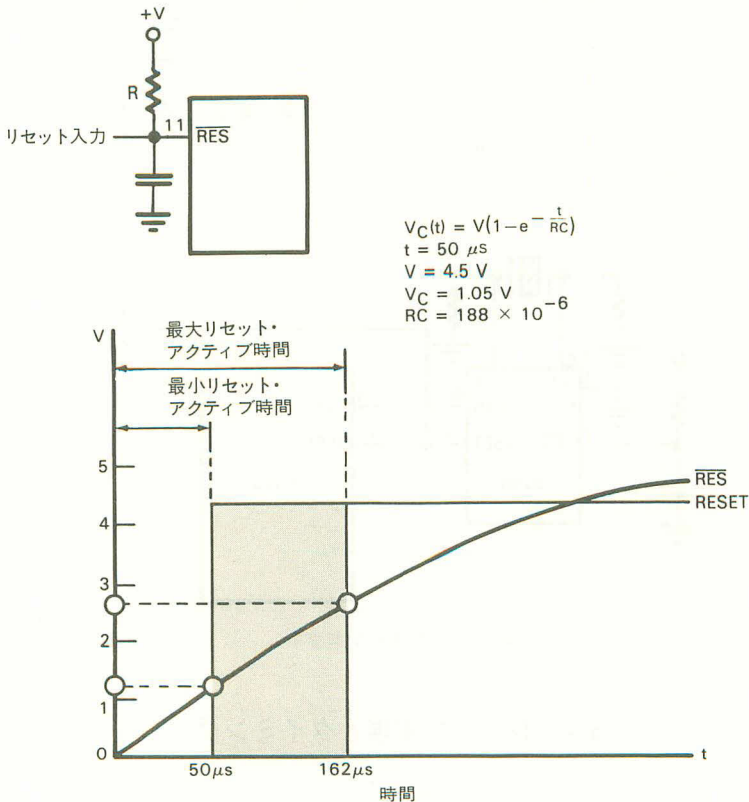



図8-19　8284のリセット回路

計算されたＲＣ値は，電源供給が4.5ボルトに達する時間あるいはこの期間に蓄積される電荷を含んでいる．ヒステリシスがなければ，入力電圧が入力のスイッチング電圧を通過するのに従って，リセット出力は振動する．計算されたＲＣ値は，1.05ボルトのレベルでスイッチする8284の50マイクロ秒のリセット期間と，2.6ボルトのレベルでスイッチする82

84の約162マイクロ秒のリセット期間に必要な最小値を与える．最小と最大のリセット時間の間により小さい許容差が必要ならば，簡単なＲＣ回路よりも，**図8-20**に示されているリセット回路が用いられる．

図8-20に示す回路は，ＲＣ回路の逆指数関数的な充電速度ではなく，定電流源でキャパシタに対する線形充電速度を与える．この回路の最大リセット期間は124マイクロ秒である．

図8-21に示すように，ＣＰＵに対するリセット信号を発生するために，8284はリセット入力とＣＰＵクロックの同期をとる．

出力はまたシステム全体の全般的なリセットとして用いられる．リセットは，8284のクロック回路には何の影響も与えない．

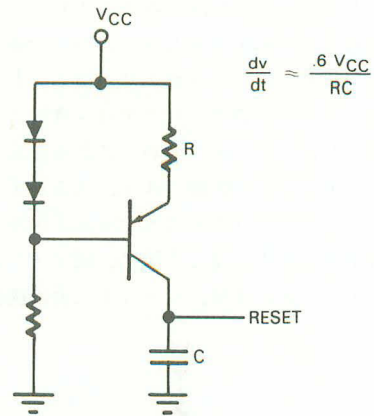



図8-20　定電流パワーオン・リセット回路

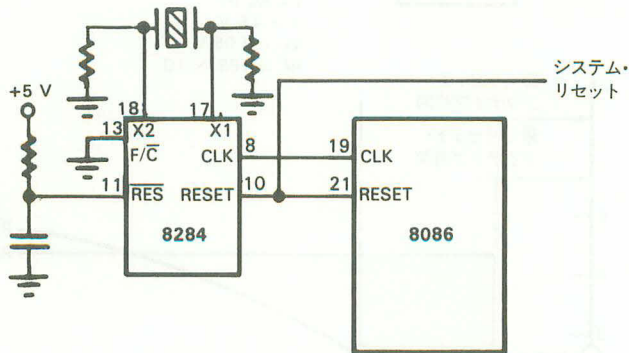



図8-21　8086のリセット

8.4　レディの実現とタイミング

最大のＣＰＵバス帯域幅で情報を伝送できないメモリとI/O素子を適応させるために，8086はREADY信号を用いる．READYはまた，マルチマイクロプロセッサ・システムで8086のシステム・バスに対するアクセスをウエートさせるために用いられる．バス・サイクルにウエート状態を挿入するために，ＣＰＵに対するREADY信号は，T2の終わりまでにインアクティブ（ロー）でなければならない．ウエート状態の挿入を避けるためには，T3期間の正の遷移の前に指定されたセットアップ時間内に，READYはアクティブ(ハイ)でなければならない．システムの規模と特性に依存して，READYのロジックは次の２つ

の方法の1つをとる.

(1) システムは通常，ノット・レディである．選択されたメモリまたはI/O素子でデータ伝送の用意ができると，ハイのREADY信号を入力する.

(2) システムは通常，レディである．選択されたメモリまたはI/O素子がCPUの最大伝送速度でデータ伝送ができなければ，ローのREADY信号を入力しなければならない.

“正統派的”なREADYの実現は，システムを“通常はノット・レディ”の状態に保つ．選択された素子が読み込み/書き込み，またはインタラプト・アクノリッジのコマンドを受け取ると，このコマンドに応答するのに十分な時間があれば，8086にREADYのハイを入力する．これにより，8086はバス・サイクルを進めることができる.

この実現は，規模の大きいマルチマイクロプロセッサ，マルチバス・システム，あるいは伝播遅延，バス・アクセス遅延，さらに素子の特性が固有にシステムを遅くする場合の特徴となる．この方法を用いて，ウエート状態なしで動作できる素子は，最大のシステム性能に対して前述の制限以内にREADYのハイを返さなければならない．高速の素子が時間内に応答できなければ，バス・サイクルにウエートのクロック期間が挿入される.

代わりの方法は，“通常はレディ”のシステムを持つことである．すべての素子は，最大のCPUバス帯域幅で動作することを仮定している．必要条件を満たさない素子は，ウエート状態のクロック期間の挿入を確実にするために，T2の終わりまでにREADYのローを入力しなければならない．この実現は一般に，規模の小さいシングルCPUシステムに適用される．これにより，READY信号のコントロールに必要なロジックが減少する．ウエート状態を要する素子がT2の終わりまでにREADYをディスエーブルにできなければ，バス・サイクルを早まって終結させることになるので，この方法を用いるときは，システムのタイミングを十分に解析しなければならない.

8086システムでは，システムの性能の最適化のために，シングル・システムにおいて前述の2つのREADYの方法を設計者が結合できることを10章に示す.

8086は，システムの実現に依存して，READYに対する2つの異なるタイミング条件を持つ．“通常はノット・レディ”のシステムでは，ウエート状態を避けるために，T3の間の

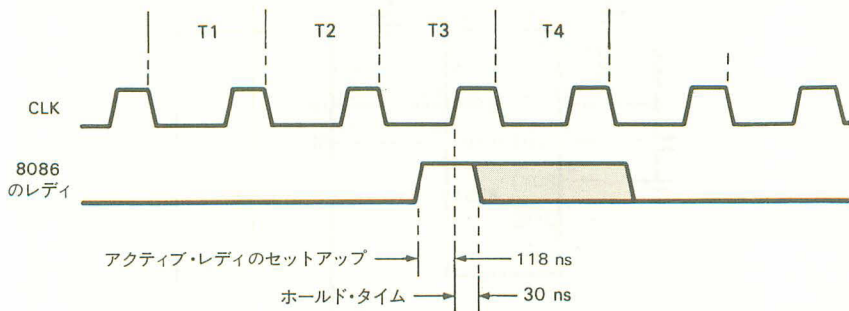



図8-22 ウエート状態を避ける通常ノット・レディのシステム

正のクロック遷移の118ns (TRYHCH) 以内に READY はハイでなければならない．この様子を**図8-22**に示す．

“通常はレディ”のシステムは，**図8-23**に示すように，T2の終わり(T3の初め)の後の8ns (TRYLCL) 以内に READY のローを入力して，ウエート状態を挿入しなければならない．

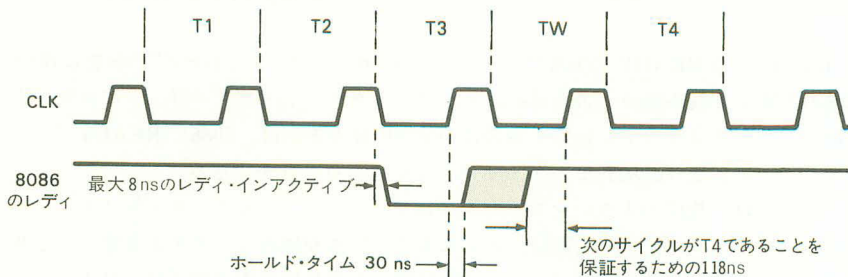



図8-23 ウエート状態を挿入する通常レディのシステム

8086の正しい動作を保証するために，T3 のクロックがローの期間は，READY 入力をハイからローに変化させてはならない．両方の場合において，READY は，T3 の正のクロックの遷移から30ns(TCHRYX) のホールド・タイムを満たさなければならない．

前述のセットアップとホールド・タイムを満たす安定な READY 信号を得るために，8284は2つの分離したシステム・レディ入力 (RDY1, RDY2) と1つの同期レディ出力(READY)を備えている．RDY入力は別々のアクセス・イネーブル ($\overline{\text{AEN1}}$, $\overline{\text{AEN2}}$)とのゲートを持つ．これにより，**図8-24**に示すように，2つの READY 信号から1つを選択することができる．ゲートを持つRDY信号は論理的に8284によってORがとられ，CPUに対する READY を発生するために，各CLKサイクルの開始でサンプルされる．このタイミングを**図8-25**に示す．

サンプルされた READY 信号は，“ノット・レディ”と“レディ”についてのCPUのタイ

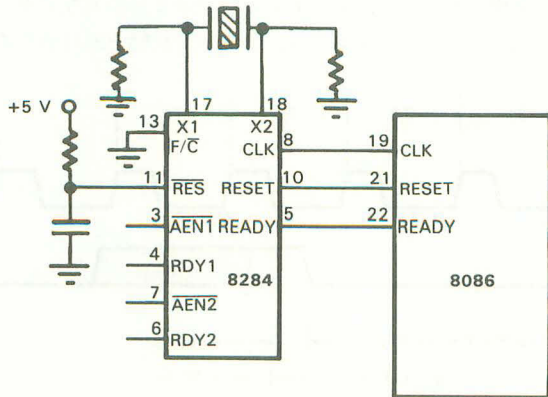



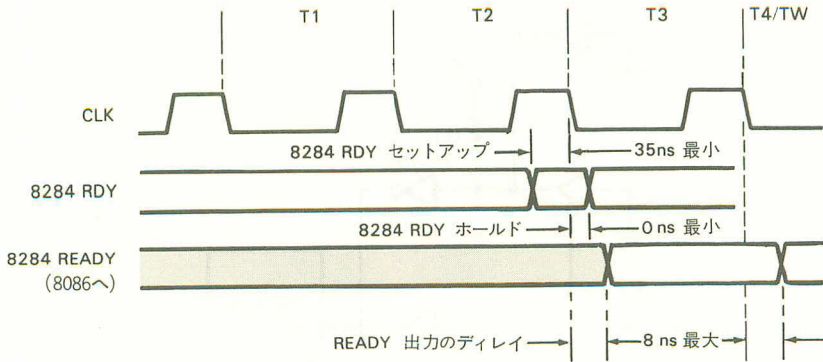
図8-24 8284と8086のレディの接続

注）8284のデータ・シートには，レディ出力のディレイ（TRYLCL）は，T2の終わりの前－8nsと記されている．これは図に示されているタイミングを意味する．

図8-25　8284と8086のレディのタイミング

ミング条件を満たすために，ＣＬＫの後の8ns（TRYLCL）以内は有効である．READYは次のＣＬＫまで変化できないので，ホールド・タイムの条件も満たされる．8284に対するシステムのレディ入力（RDY1，RDY2）はT3より35ns（TRIVCL）前で有効で，$\overline{\text{AEN}}$はT3より60ns前で有効でなければならない．１つのＲＤＹ入力のみを用いたシステムでは，**図8-26**に示すように，関連のある$\overline{\text{AEN}}$はグランドに接続され，もう１つの$\overline{\text{AEN}}$は１キロオームを通して５ボルトに接続される．

システムがロー・アクティブのレディ信号を生成すると，8284の$\overline{\text{AEN}}$入力によって必要とされる付加的なセットアップ・タイムが満たされれば，それは8284の$\overline{\text{AEN}}$入力に接続することができる．この場合，**図8-27**に示すように，関連のあるＲＤＹ入力はハイに接続される．

ＣＰＵの最大周波数以下で動作するメモリと周辺素子の大多数は一般に，１つ以上のウエート状態を必要としない．**図8-28**の回路は１つのウエート状態を発生する．

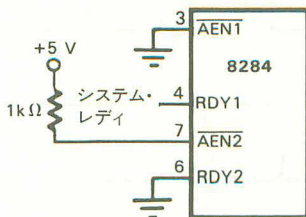



図8-26　１つのRDY入力を用いた8284

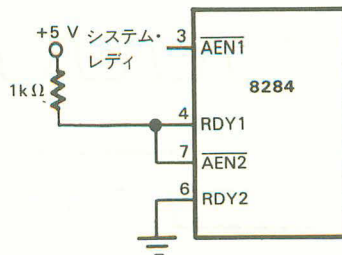



図8-27　アクセス・イネーブル駆動のシステム・レディを有する8284

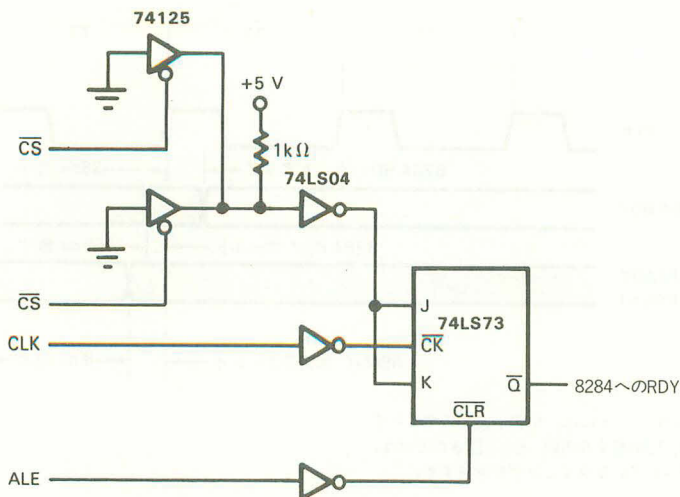



図8-28　シングル・ウエート状態の発生回路

図8-28のシステム・レディ・ラインは，1つのウエート状態を要する素子が選択されると常にローに駆動される．フリップフロップはALEでクリヤされ，8284に対するRDYをイネーブルとする．ウエート状態が必要でなければ，フリップフロップは変化しない．システム・レディがローに駆動されると，フリップフロップはT2のローからハイへのクロック変化で反転して，1つのウエート状態を強制する．CLKの次のローからハイへの変化でフリップフロップは再び反転し，レディを示してバス・サイクルの終了を可能にする．フリップフロップの状態のこれ以上の変化はバス・サイクルに影響しない．**図8-29**に示すように，回路はシステム・レディに対するチップ・セレクトに約100nsの余裕がある．

システムが“通常はノット・レディ”ならば，物理的なメモリの最後の6バイトにプログラムは実行可能コードを割り当てられない．8086は命令を前もってフェッチするので，物

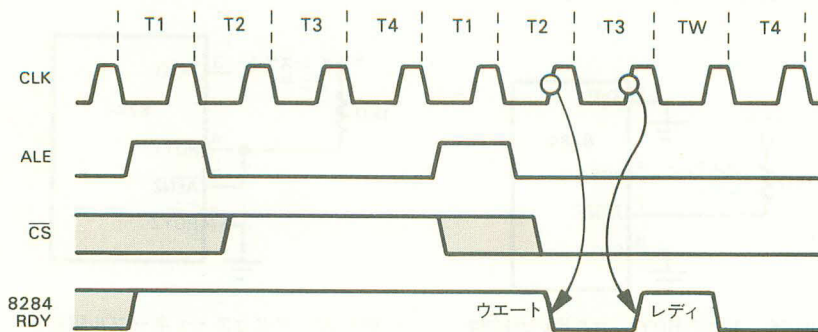



図8-29　シングル・ウエート状態の発生回路のタイミング

理的なメモリの終わりのコードを実行するときは，ＣＰＵは存在しないメモリをアクセスしようとする．存在しないメモリに対するアクセスがREADYをイネーブルにできなければ，システムは不定のウエート状態に陥る．

8.5　インタラプト構造

8086のインタラプト構造は，**図8-30**に示されているように，メモリ位置0_{16}から$003FF_{16}$までにストアされるインタラプト・ベクタのテーブルに基づいている．各ベクタは4バイ

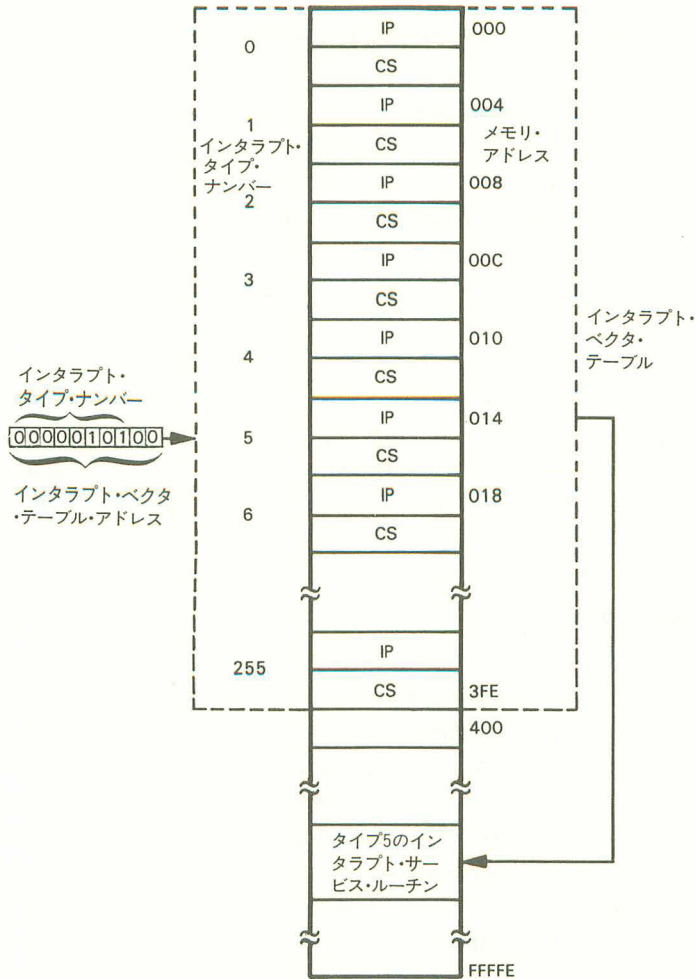



図8-30　インタラプト・ベクタ・テーブルからのインタラプト・サービス・ルーチン・アドレスの獲得

トで，最初の2バイトは新しいプログラム・カウンタ・アドレスを，次の2バイトは新しいコード・セグメント・レジスタ・アドレスを保持している．

この2つのアドレスは結合されて，インタラプト・サービス・ルーチンの20ビットの実行アドレスを形成する．この20ビットのアドレスは，通常の8086のセグメント・プログラム・メモリ・アドレッシングを用いて計算される．インタラプト・ベクタ・テーブルは，8086の1メガバイトのアドレス空間の任意の場所に位置するインタラプト・サービス・ルーチンの開始アドレスを指定する，256までのインタラプト・ベクタを含む．

特定の構成で256以下のインタラプトを用いるならば,用いられているインタラプト・ベクタのメモリを割り当てるだけでよい．しかしシステムのデバッグを行なうときは，誤った割り込みを検出する手段として，すべての未定義の割り込みをトラップ・ルーチンに割り当てなければならない．

各インタラプト・ベクタは，関連のあるインタラプト・ナンバーを持つ．インタラプト・ナンバーは，インタラプト・ベクタ・テーブル内のインタラプト・ベクタを識別する．インタラプト・ナンバーに4を乗じることによって，インタラプト・ベクタ・テーブル内のインタラプト・ベクタのエントリの最初のバイトに対する絶対アドレスが得られる．たとえば，インタラプト・ナンバー5は，インタラプト・ベクタ・テーブルの6番目のエントリを示し，このベクタの最初のバイトはアドレスが20_{10}（$=14_{16}$）である．この様子を図8-30に示す．

このようにして，8086のインタラプト構造により，各インタラプト・サービス・ルーチンに対する開始メモリ・アドレスが指定できる．

8086で特殊な機能によって要求される定義済のインタラプト，ユーザ定義のハードウエア・インタラプト,ソフトウエア・インタラプトの3つの型のインタラプトが8086にはある．

定義済インタラプトは，ハードウエアやソフトウエアによって要求することができる．次に定義済インタラプトについて詳細に検討する．

8.5.1 定義済インタラプト

“定義済”インタラプトは，割り当てられたインタラプト・ナンバーと自動的にベクタを決定するロジックを持つことからそう呼ばれる．したがって，定義済インタラプトが受け付けられると，8086のロジックは自動的に，インタラプトの割り付けられたベクタ・テーブル・エントリのベクタ決定を行なう．ただし，プログラム・カウンタとコード・セグメントのアドレスでベクタ・テーブル・エントリを初期設定して，各インタラプトにインタラプト・サービス・ルーチンを用意しなければならない．

外部ロジックによって要求される定義済ハードウエア・インタラプトと，命令実行の結果として要求される定義済ソフトウエア・インタラプトが存在する．

インタラプト・ナンバー0から31までは，定義済インタラプトに割り当てられている．定義済インタラプトを用いなければ，そのインタラプト・ナンバーを何か他のインタラプトに用いることができる．しかし，このようなシステムは，将来における8086のハードウエアとソフトウエアの製品との互換性がなくなるので，推奨されない．

次に定義済インタラプトを1つずつ述べる.

(1) インタラプト　0　—　0による除算

このインタラプトは，除算命令の実行に続いて，商が除算命令で許される最大値を越えると自動的に要求される.このインタラプトはマスクすることができない.これは標準の除算命令実行ロジックの一部として要求される. 0による除算のインタラプト・サービス・ルーチンによってインタラプトが再びイネーブルにされなければ，“ワースト・ケース”の除算命令時間の計算に，このサービス・ルーチンの実行時間が含まれる. これは除算命令に対して最も長い実行時間となる.

(2) インタラプト　1　—　シングル・ステップ

このインタラプトは，プログラム・ステータス・ワードでTF(トラップ・フラグ) がセットされて1命令後に発生する. このインタラプトは，プログラムを一度に1命令実行させるために用いられる. プログラムの各命令が実行された後で，インタラプトが要求される. インタラプトの要求に続いて，次に実行されるプログラムの命令の結果に，インタラプト・サービス・ルーチンによって種々の診断機能が行なわれ，そして次のシングル・ステップのインタラプト要求が発生する.

シングル・ステップの開始には，プログラム・ステータス・ワードの内容をスタックにプッシュして，スタックのトップにセーブされたプログラム・ステータス・ワード内のトラップ・フラグをセットして，プログラム・ステータス・ワードにスタックからポップする. シングル・ステップのインタラプトは，次の命令の実行に続いて要求される.

シングル・ステップのインタラプト要求が受け付けられると，シングル・ステップのインタラプト・サービス・ルーチン自身がシングル・ステップのインタラプト要求によるインタラプトを受けないように，プログラム・ステータス・ワード中のTFフラグはリセットされる. スタックにセーブされているフラグのTFはセットされたままである.

シングル・ステップのインタラプト・サービス・ルーチンからの復帰にはIRET命令を用いなければならない. この復帰はフラグ（TFを含む）を復元して，別のTFインタラプトが次の命令の終了で発生することを可能にする.

(3) インタラプト　2　—　NMI(ノンマスカブル・インタラプト)

これは最も高い優先度のハードウエア・インタラプトである. その名前が意味しているように，マスクすることはできない. NMIインタラプト要求の入力は，NMI入力のローからハイへの遷移によるエッジ・トリガであり，内部的にCPUのクロック信号CLKのローからハイへの遷移と同期している. したがってNMIは，認識を保証するために少なくとも2クロック期間，ハイでなければならない. NMI入力のローからハイへの遷移でインタラプト要求を生成できるので，擬似的な遷移は抑圧する必要がある.

NMIが通常ハイならば，認識を保証するためにアクティブなローからハイへの遷移を行なう前に，2つのCPUクロック期間はローでなければならない. この入力は一般に，たとえばパワー低下やシステムのウォッチドッグ・タイマの時間超過に続く，非常時のインタラプト要求のために保留されている.

(4) インタラプト　3　—　1バイト・インタラプト

これはソフトウエア・インタラプトである．これは，1バイトのオブジェクト・コードを占める特殊なインタラプト要求命令の実行によって発生する．これは，ソフトウエア・デバッグ・プログラムでブレイクポイントを設定するために用いられる．最小の8086命令オブジェクト・コードは1バイトなので，1バイト・インタラプトはブレイクポイントを設定する手段としてどの8086命令とも置き換えられる．

この1バイト・インタラプトはマスクできない．

(5) インタラプト　4　—　インタラプト・オン・オーバーフロー

このインタラプト要求は，プログラム・ステータス・ワードのオーバーフロー・フラグ（ＯＦ）がセットされて，INTO命令が実行されると発生する．INTO命令により，8086のオーバーフロー・エラー・サービス・ルーチンへのトラップが可能になる．インタラプト・オン・オーバーフローはマスクできない．

8.5.2 ユーザ定義ソフトウエア・インタラプト

2バイト・インタラプトの INT nn 命令の実行によって，ソフトウエア・インタラプトを起こさせることができる．最初のオブジェクト・コード・バイトはＩＮＴのオペコードで，次のオブジェクト・コード・バイト(nn)は実行されるべきインタラプトのナンバーを含む．ＩＮＴ命令はマスクできない．

この命令は，ダイナミックにリロケート可能なプログラムを呼び出すためによく用いられ，メモリでの呼び出されたプログラムの位置は呼び出し元プログラムに知られていない．ただし，呼び出されるプログラムがメモリにロードされると，その実行アドレスは，インタラプト・ベクタにロードされる．呼び出されたプログラムは，インタラプト・リターン(IRET) 命令で復帰しなければならない．

8.5.3 ユーザ定義ハードウエア・インタラプト

マスク可能なハードウエア・インタラプトは8086の INTR ピンによって要求され，このインタラプトはプログラム・ステータス・ワードのＩＦビット（インタラプト・フラグ）によってマスクできる．各命令実行の最後のクロック期間に，INTR ピンの状態がサンプルされる．この規則には次の2つの例外がある．

1．命令がセグメント・レジスタへのＭＯＶあるいはセグメント・レジスタへのＰＯＰであるとき．

2．それが先行する命令の一部として取り扱われる命令のプレフィックスの実行の間．

この2つの例外は，“一般的な場合”のインタラプト・アクノリッジ実行シーケンスの記述に続いて述べる．

8.5.4 インタラプト・アクノリッジ・シーケンス

“一般的な場合”としてユーザ定義のハードウエア・インタラプトを取り上げて，インタラプト・アクノリッジ・シーケンスについて述べる．

サンプル時にINTR信号がハイでプログラム・ステータス・ワードのIFビットが1ならば，ユーザ定義のインタラプトは要求される．このインタラプトはイネーブルなので，8086はインタラプト・アクノリッジ・シーケンスを実行する．インタラプトが受け付けられるのを保証するために，ミニマム・システムでは$\overline{\text{INTA}}$によって，マキシマム・システムでは$\overline{\text{S0}}$, $\overline{\text{S1}}$, $\overline{\text{S2}}$によって，8086がインタラプト・アクノリッジを返すまで，INTR 入力はハイに保持されなければならない．

現在の命令が実行を終了するときにBIUが命令をフェッチしているならば生じるような，BIUがインタラプト条件の検出されるバス・サイクルならば，INTR のインタラプト・リクエストは，バス・サイクルのT4より前の2クロック期間は有効でなければならない．そうでなければ，インタラプト・アクノリッジが発行される前に(1つは保留でも)別のバス・サイクルが実行される．

ロックされた命令の実行中にインタラプトとホールドが要求されたならば発生するように，ホールド・リクエストが保留されていれば，ホールドが最初に処理され，ホールドの処理が終わってからインタラプトが受け付けられる．

INTR ピンに生じるユーザ定義ハードウエア・インタラプトの要求だけが特定のハードウエア・アクノリッジを受け取る．このアクノリッジは，**図8-31**に示されているように，2つのアイドル・クロック期間によって分離された，2つのインタラプト・アクノリッジ・バス・サイクルの形をとる．ソフトウエア・インタラプトとノンマスカブル・インタラプトは，図8-31に示されているアクノリッジ・シーケンスを受け取らない．

図8-31に示されているように，完全なインタラプト・アクノリッジ・シーケンスは，2つのアイドル・クロック期間で分離された，2つの$\overline{\text{INTA}}$バス・サイクルから成る．2つのバス・サイクルの間，$\overline{\text{INTA}}$はインタラプトのアクノリッジのために（ミニマム・モードでは）ローが出力される．アドレス/データ・バス（$\overline{\text{BHE}}$を含む）と関連のあるステータス（S3－S7）は2つのバス・サイクルの間，フロート状態となる．しかし，ハイのAL

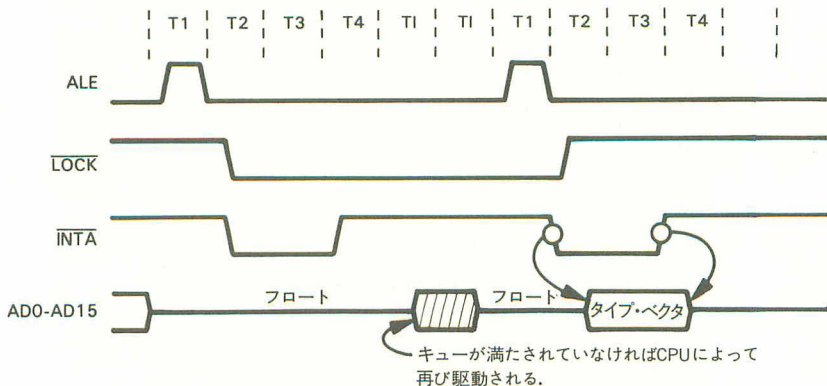



図8-31　ミニマム・モードにおけるインタラプト・アクノリッジ・シーケンス

Eパルスが出力されるので，アドレス・ラッチには不確定の情報がロードされる．したがって素子は，出力を駆動する前に制限として，常にREAD($\overline{\mathrm{RD}}$)のローを用いるべきである．

$\overline{\mathrm{INTA}}$ バス・サイクルの間，DT/$\overline{\mathrm{R}}$ と $\overline{\mathrm{DEN}}$ はアクティブである．これにより，インタラプトを要求する素子からの1バイトのインタラプト・ナンバーを8086が受け取ることができる．

最初のバス・サイクルはインタラプト・アクノリッジが進行中であることを知らせる．これにより，次の $\overline{\mathrm{INTA}}$ バス・サイクルの間に伝送するためのインタラプト・ナンバーを用意する時間を，インタラプトを発行した素子に可能とする．インタラプト・ナンバーは，第2の $\overline{\mathrm{INTA}}$ バス・サイクルの間に16ビット・データ・バスの下位で8086に伝送されなければならない．したがって，インタラプト・ベクタを与える素子は，16ビット・データ・バスの下位に接続しなければならない．

$\overline{\mathrm{INTA}}$ バス・サイクルのタイミング（アドレスのタイミングを除く）は，リード・バス・サイクルのタイミングと同様である．

8086のインタラプト・アクノリッジ・シーケンスは，インタラプト・アクノリッジ・シーケンスの間にＣＰＵによって読み出される命令のない 8080 や 8085 とは異なっていることに注意．8080と8085は，アクノリッジ・シーケンスの一部としてＣＰＵに対してインタラプトを発行した素子によって出されるリスタートあるいはコールの命令を必要とする．

ミニマム・モード・システムにおいて，インタラプト・アクノリッジ・バス・サイクルの間，M/$\overline{\mathrm{IO}}$信号はローである．

8086は，ＢＩＵが2つのINTAサイクルの間に発生するホールド・リクエストを受け付けるのを防止している．

マキシマム・モード・システムでは，ステータス・ライン $\overline{\mathrm{S0}}$—$\overline{\mathrm{S2}}$ により，8288バス・コントローラはインタラプト・アクノリッジ・バス・サイクルの間，$\overline{\mathrm{INTA}}$ のローを出力する．どちらの $\overline{\mathrm{RQ}}$/$\overline{\mathrm{GT}}$ 入力におけるホールド・リクエストも8086が受け付けるのを防止し，マルチマスタ・システムにおいて2つのインタラプト・アクノリッジ・バス・サイクルの間にバス判定ロジックがバスを放棄するのを防止するために，最初のインタラプト・アクノリッジ・バス・サイクルのT2から，次のインタラプト・アクノリッジ・バス・サイクルのT2まで，8086の $\overline{\mathrm{LOCK}}$ 出力はアクティブとなる．

8086は，セグメント・レジスタに対するＭＯＶあるいはセグメント・レジスタへのポップの次にはINTRをサンプルしない．これにより，2つのロードをインタラプトが分離する可能性なしに，スタック・ポインタＳＳとＳＰのレジスタへの32ビット・ポインタのロードを可能にする．

次はインタラプトの起きない命令列の例である．

```
MOV    SS，NEW$STACK$SEGMENT
MOV    SP，NEW$STACK$POINTER
```

プレフィックスはそれが先行する命令の一部として取り扱われるので，プレフィックス命令の実行後に8086はINTRをサンプルしない．この規則の1つの例外は，ストリング・プリミティブにリピート（ＲＥＰ）プレフィックスが先行するときに起きる．繰返しのスト

リング操作は，各々の繰返しのストリング・プリミティブの実行が終了するごとにINTRをサンプルする．これには，LOCKフレフィックスを持つ繰返しのストリング操作が含まれる．複数のフレフィックスが繰返しのストリング操作に先行し，命令がインタラフトで中断されると，ストリング・フリミティブ直前のフレフィックスだけが，インタラフト・ルーチンからの復帰に続いて回復させられる．フログラム実行の正しい再開を可能とするためには，次のプログラミング技法を用いなければならない．

```
LOCKED$BLOCK$MOVE:      LOCK
                        REP
                        MOVS      DEST,CS: SOURCE
                        AND       CX,CX
                        JNZ       LOCKED$BLOCK$MOVE
```

MOVS命令のために生成されるオブジェクト・コード・バイトは，（下降順に）LOCKフレフィックス，REPフレフィックス，セグメント変更フレフィックス，そしてMOVSである．インタラフトから復帰すると，セグメント変更フレフィックスは，正しいメモリ位置の間で1つの付加的な伝送が起きることを保証するために回復させられる．移動に続く命令は，移動が完了したかを判断するために繰返しのカウント値を調べる．移動が完了していなければ，ブロック移動命令への復帰が起きる．

8086は，ハードウエア・インタラプトについてはバスから，ソフトウエア・インタラフトについては命令の流れから，インタラプト・ナンバーを読み込む．インタラプト・ナンバーは，インタラフト・ベクタ・テーブル中の対応するインタラフト・ベクタのアドレスを生成するために，4が乗じられる．インタラプト・ベクタの4バイトは次のものである．

- プログラム・カウンタの下位バイト
- プログラム・カウンタの上位バイト
- コード・セグメント・レジスタの下位バイト
- コード・セグメント・レジスタの上位バイト

次に，8086はフログラム・ステータス・ワードの内容をスタックにプッシュし，トラップとインタラプトのフラグをリセットして，それから現在のコード・セグメント・レジスタとプログラム・カウンタの内容をスタックにプッシュする．新しいコード・セグメント・レジスタとフログラム・カウンタの内容がインタラプト・ベクタ・テーブルからロードされる．すなわち，このためにリード・バス・サイクルが実行される．

インタラプト・アクノリッジ・シーケンスの間のインタラフト・ベクタ・テーブル参照では，セグメント・レジスタは用いられない．20ビットのアドレスを形成するために，ベクタのディスプレイスメントが0に加えられ，セグメント・レジスタが選択されていないことを示すために，S4は1でS3は0となる．

以下は，ユーザ定義のマスク可能なインタラプトが受け付けられたときに実行される実際のバス・シーケンスである．

1．2つのアイドル・クロック期間で分離された，2つのインタラプト・アクノリッジ・バス・サイクルが実行される．図8-31に示しているように，受け付けられた素子は，

第2のインタラプト・アクノリッジ・バス・サイクルの間に1バイトのデータとしてインタラプト・ナンバーを返す．このデータ・バイトは，2ビット左へシフトされて，インタラプト・ベクタの開始アドレスになる．

2．リード・バス・サイクルが実行され，この間に新しいCSレジスタの内容がインタラプト・ベクタの最初の2バイトから読み出される．

3．リード・バス・サイクルが実行され，この間に新しいプログラム・カウンタの内容がインタラプト・ベクタの第3と第4のバイトから読み出される．

4．ライト・バス・サイクルが実行され，この間にプログラム・ステータス・ワードの内容がスタックにプッシュされる．

5．プログラム・ステータス・ワードのインタラプト(IF)とトラップ(TF)のフラグが0にリセットされる．これにより，マスク可能なものあるいはシングル・ステップのインタラプトがディスエーブルとなる．

6．ライト・バス・サイクルが実行され，この間にCSレジスタの内容がスタックにプッシュされる．

7．ライト・バス・サイクルが実行され，この間にプログラム・カウンタの内容がスタックにプッシュされる．

次にプログラムの実行は，そのアドレスがインタラプト・ベクタからフェッチされている，インタラプト・サービス・ルーチンに分岐する．

ノンマスカブル・インタラプト，ソフトウエア・インタラプト，あるいはシングル・ステップ・インタラプトが受け付けられたときは，前記のステップ2から7が実行され，インタラプト・ナンバーは既知なのでステップ1は必要でない．

ユーザ定義のマスク可能なインタラプトが要求されたときの命令の終わりと，インタラプト・サービス・ルーチン実行の始まりとは，62クロック期間で分離されている．

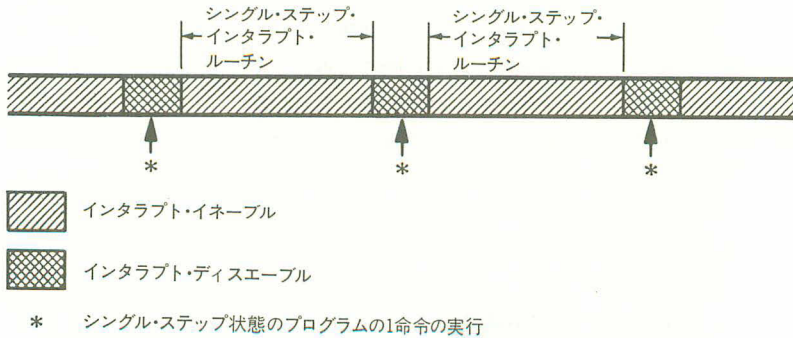
ソフトウエア生成のインタラプトに対しては，インタラプト・アクノリッジ・バス・サイクルが実行されないことを除いて，同じシーケンスのバス・サイクルが実行される．結果として，インタラプト・サービス・ルーチンの実行に対するディレイは，INT nn とシングル・ステップで51クロック期間，INT3 で52クロック期間，そして INTO で53クロック期間となる．

バス・サイクルにウエート状態が挿入されれば，前記のインタラプト・アクノリッジ・クロック期間の数は当然，それに応じて増加する．

次に，複数インタラプトとインタラプトの優先権について検討する．

INTR による外部インタラプト要求のみがディスエーブルとすることができる．結果的に，このインタラプトは最も低い優先権を持つ．他のインタラプトのアクノリッジ・シーケンスは，プログラム・ステータス・ワードのIFフラグをリセットする．したがって，INTR によって要求されたインタラプトは，他のインタラプトのサービス・ルーチンが完了するか，あるいはインタラプトが再びイネーブルとなる（IFフラグがセットされる）まで，受け付けることができない．

シングル・ステップを用いてデバッグを行なっているプログラムは，シングル・ステッ

プ・インタラプト・サービス・ルーチン内だけで，外部のユーザ定義インタラプトを受け付けるように変更することができる．これにより，外部インタラプトをシングル・ステップにもかかわらず短時間で処理できる．このためには，シングル・ステップ・インタラプト・サービス・ルーチンが，スタックのトップの2バイトにストアされているプログラム・ステータス・ワードにある，割り込まれたプログラムのＩＦフラグをリセットし，シングル・ステップ・ルーチンでのインタラプトをイネーブルにする必要がある．これは次のように図示される．

一方，割り込まれたプログラムだけ，あるいは外部のユーザ定義インタラプトのサービス・ルーチンだけのシングル・ステップが必要となる場合がある．割り込まれるプログラムによってプログラム・ステータス・ワードのＴＦフラグが1にセットされれば，割り込まれるプログラムはシングル・ステップとなり，そうでなければシングル・ステップにはならない．どちらの場合も，ユーザ定義インタラプトのサービス・ルーチンはＴＦを0にリセットして実行を開始し，したがってシングル・ステップは無効となる．インタラプト・サービス・ルーチンにおけるプログラム・ロジックはその結果，インタラプト・サービス・ルーチンの実行の間，シングル・ステップをイネーブルにしなければならない．

必要ならば，シングル・ステップのトラップでINTR をディスエーブルにできる．これには，シングル・ステップ・インタラプト・サービス・ルーチンがプログラム・ステータス・ワードのＩＦフラグを0にリセットしておくことが必要である．長時間にもわたってINTR をディスエーブルにするパスは，プログラム・ロジックの分裂を引き起こす可能性がある．

次に，マスク不可能なインタラプトの優先権について検討する．ＮＭＩ，シングル・ステップ，そしてソフトウエアのトラップの3つのインタラプトについては既に述べた．すべてINTR によるユーザ定義の外部インタラプト要求より高い優先権を持つ．これらの中で，3つのノンマスカブル・インタラプトの2つが同時に起きると，シングル・ステップが最も高い優先権を持ち，次にＮＭＩが続き，ソフトウエア・トラップは最も低い優先権を持つ．しかし3つのノンマスカブル・インタラプトすべてが同時に要求されると，ＮＭＩが最も優先権が高く，次いでソフトウエア・トラップが続き，シングル・ステップの優

先権が最も低くなる.

シングルステップは，優先権がＮＭＩより高いかあるいは低いので，シングル・ステップ・インタラプト・サービス・ルーチンはその実行がＮＭＩインタラプトに続いているかどうかを調べる必要がある．もしＮＭＩインタラプトに続いていて，直ちにＮＭＩの処理が必要ならば，シングル・ステップ・インタラプト・サービス・ルーチンは自分自身をディスエーブルにするロジックを含まなければならない．このプログラム・ロジックは，スタックのトップのリターン・アドレスを調べ，ＮＭＩインタラプト・サービス・ルーチンのアドレスを検出すると，ＮＭＩルーチンの実行を可能とするために復帰だけを行なう必要がある．ＮＭＩルーチンはシングル・ステップ状態のプログラムに復帰し，シングル・ステップは復帰の間にフラグが再現されて自動的に再びイネーブルとなる．実際の影響は，ＮＭＩが検出されると，シングル・ステップ状態のプログラムの１命令に対してシングル・ステップが無視されることである．シングル・ステップはインタラプト・アクノリッジの過程でディスエーブルなので，ＮＭＩインタラプト・サービス・ルーチンはその実行の間，シングル・ステップをディスエーブルとするために，プログラム・ステータス・ワードのＴＦフラグをリセットしておくことだけが必要である.

8.5.5　システムのインタラプト構成

8259A プライオリティ・インタラプト・コントローラは，INTR を通して要求される複数の外部のユーザ定義インタラプトを取り扱うことができる．この素子は，8080A／8085あるいは8086システムで動作する．8259A は縦続接続が可能で，マスタ／スレーブ構成で，１つのシステムで64のインタラプトまで取り扱うことができる.

図8-32と**図8-33**に，ミニマム・モードとマキシマム・モードの8086システムにおける8259A を示す.

図8-32a に示すミニマム・モード構成は，8086の多重化バスに接続されている8259A を示す．図8-32b に示す構成は，分離バス・システムに接続されている8259A を示す．これらの相互接続はまた，マキシマム・モード・システムにも適用できる．マキシマム・モード・システムに取り上げた構成は，付加のスレーブ8259A がバッファを用いたシステム・バス上にあり，8086多重化バス上のマスタ8259A を示している．この構成は，マキシマム・モード・システム・インターフェイスのいくつかの独特な特徴を表わしている．マスタ8259A がスレーブ8259A と正規の割り込みの素子の混合したインタラプトを受けると，スレーブは接続されている素子のインタラプト・ナンバーを供給する必要があり，一方マスタはそのインタラプト入力に直接所属しているインタラプト・ナンバーを供給しなければならない.

マスタ8259A は，インタラプトが要求のある素子から直接かあるいはスレーブ8259A から受けているものかを判断することができる．マスタ8259A は，この情報をデータ・バス・トランシーバ（$\overline{DEN}$ と $\overline{EN}$ の NAND 機能によって）をイネーブルあるいはディスエーブルとするために用いる．マスタ8259A がインタラプト・ナンバーを供給しなければならないならば，データ・バス・トランシーバをディスエーブルにする．スレーブ8259A がタイ

図8-32(a)　ミニマム・モード8086に接続された8259 — 多重化バス

プ・ナンバーを供給しなければならないならば，8086はデータ・バス・トランシーバをイネーブルにする．

$\overline{EN}$出力は通常ハイで，8086／8288がバス・トランシーバをコントロールすることを可能にしている．スレーブ・インタラプト処理時の適当なスレーブを選択するために，マス

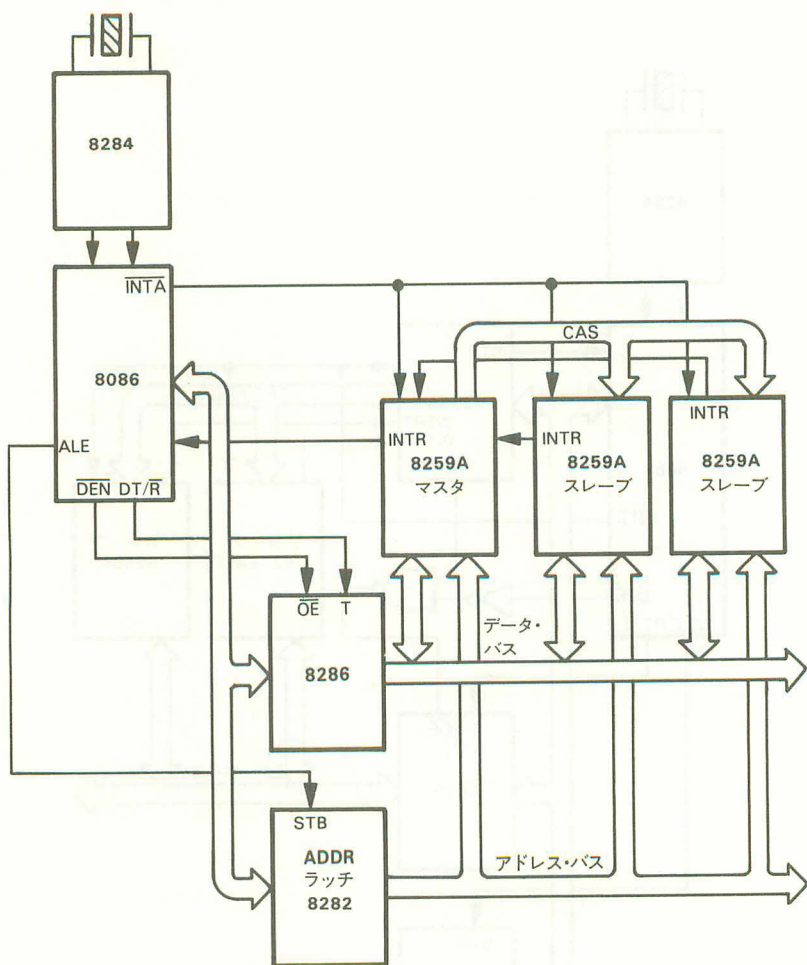



図8-32(b)　ミニマム・モード8086に接続された8259 — 分離バス

タはスレーブに対してカスケード・アドレス（CAS）を与えなければならない．8288がI/Oバス・モードに指定されていない（8288のIOB入力がグランドに接続されている）ならば，MCE/$\overline{\text{PDEN}}$出力はMCEすなわちマスタ・カスケード・イネーブル出力となる（I/Oバス・モードの利用は10章に説明されている）．**図8-34**に示されているように，MCEは$\overline{\text{INTA}}$サイクルの間だけアクティブである．MCEはALEの間，8086のローカル・バス上にマスタ8259Aのカスケード・アドレスをイネーブルとする．

これにより，用いられているシステム・アドレス・バスが適当なスレーブ8259Aを選択して，アドレス・ラッチがALEによってカスケード・アドレスを捕えることが可能にな

図8-33　マキシマム・モード8086に接続された8259

る．8086がそのバス出力をフロート状態にし，カスケード・アドレス（CAS）がバス上でイネーブルとなるまでの間のローカル・バスの競合を最小にするために,MCE は $\overline{\text{LOCK}}$ とのゲートを有する．

最初の $\overline{\text{INTA}}$ バス・サイクルで，マスタ8259A は内部のプライオリティとカスケード・アドレス（CAS）の出力を決定することができ，CASは次の $\overline{\text{INTA}}$ バス・サイクルの間にスレーブへ転送される．8259A についての他の情報に関しては，インテルのアプリケーション・ノートAP59，あるいはAn Introduction to Microcomputers — Volume 2, Some Real Microprocessors, by A. Osborne を参照．

図8-34　8086ローカル・バス上に8259AがCASを出力するタイミング

8.6　8086バス・タイミング図の解釈

8086のミニマム・モードとマキシマム・モードのバス・タイミング図が，この本の後部のデータ・シートに示されている．このタイミング図は，次のように6つの区分に分けられる．

1．アドレスとALEのタイミング
2．リード・サイクルのタイミング
3．ライト・サイクルのタイミング
4．インタラプト・アクノリッジのタイミング
5．レディのタイミング
6．バス・コントロール移動のタイミング

信号のA．C．特性はCPUクロックに関して指定されているので，信号の大多数の間の関係は単に，信号が相対的な位置にあるクロックのエッジを識別しているクロック期間を決めて，適当な最小／最大のパラメータの値を加えるか減じることによって，推測することができる．この方法で補正できないシステム・タイミングの1つは，最小と最大のパラメータの値の間の“ワースト・ケース”の関係である（これはトラッキング関係としても知られている）．

たとえば，指定された最小と最大のターン・オンとターン・オフのディレイを持つ信号を考える．素子の特性に依存して，ワースト・ケースの解析が可能性を示したとしても，構成要素が同時に最大のターン・オンと最小のターン・オフのディレイを示す可能性はない．

この議論は，MOS素子の特性であり，したがって8086のA．C．特性に適用できる．ここで付言すると，最小と最大のディレイ・パラメータ混合のワースト・ケース解析は一般に，得られるワースト・ケースを超えていることがあげられる．したがって，より悪いワースト・ケースの値を得るために，解析による値を主観的にさらに下げることはできない．次に，トラッキング関係の影響を受けやすい8086のタイミングの特定の範囲に対するガイドラインを検討する．

8.7　ミニマム・モード・バスのタイミング

8.7.1 アドレスとALE

多重化バスから有効なアドレスを得る素子の能力を決定するために，アドレス/ＡＬＥのタイミング関係は重要である．8282と8283のラッチはＡＬＥの立下りエッジでアドレスを捕えるので，きわどいタイミングにはＡＬＥが終了するときのアドレス・ラインの状態が含まれる．パラメータ TAVAL ＝ TCLCH － 60ns は，ＡＬＥの立下りエッジの 58ns 前で，ＣＰＵでのアドレスは有効であることを保証している．

これは，8282 / 8283で必要とされるストローブの終わりに対する０のデータ・セットアップ・タイムを満たし，有効なアドレスが捕えられることを確実にしている．アドレスは TLLAX パラメータによってＡＬＥの終わりを過ぎても有効であることが保証されている．最新の可能性のあるＡＬＥによってアドレスは有効でないことを意味するように考えられるが，この仕様は TCHLL と TCLAX の間の関係を無効にする．

TLLAX のタイミングは，タイミング図でA 19-A 16 が示されているだけであるが，すべてのアドレス・バスに適用される．アドレスについての $TCLAX_{min}$ の仕様は，遅いＡＬＥによって制限されなくても，バスがフロート状態となる最も速い可能性のある時間を示している．TCLAX は，リード・サイクルの間の多重化アドレス / データ・ラインAD15-0 に適用されるだけである．

ライト・サイクルの間，多重化アドレス / データ・バスは直接にアドレスからライト・データに切り替わる．ＡＬＥに対するアドレス・ホールド・タイムは，TCLAX で指定される絶対最小値（ＡＬＥ終結が速い場合）で，TLLAX の仕様でも保証されている．読み込みと書き込みの両方の場合に対して，書き込みの場合の多重化アドレス / データ・バスと同じタイミングで，A 19-A 16 のラインは直接にアドレスからステータスに切り替わる．最小のＡＬＥパルス幅は $TLHLL_{min}$ で保証され，これは $TCLLH_{max}$ と $TCHLL_{min}$ の関係から得られる値よりも大きい．分離アドレス・バス上での有効なアドレスに対する最悪のディレイを決めるためには，次の２つの経路を考慮しなければならない．

1．有効アドレスのディレイ
2．ＡＬＥのディレイ

8282と8283はラッチを用いているので，ＡＬＥがアクティブとなるまで有効アドレスはアドレス・バスに転送されない．アドレスが有効となるまでのディレイ $TCLAV_{max}$ と ＡＬＥがアクティブとなるまでのディレイ $TCLLH_{max}$ の比較から，$TCLAV_{max}$ がワースト・ケースであることが示される．ラッチ伝播ディレイの減算によって，バス・サイクルの始まりからのワースト・ケースのアドレス・バスが有効となるまでのディレイが得られる．

8.7.2　リード・サイクルのタイミング

リード・サイクルのタイミングは次の３つの部分から成る．

1．バスの条件付け

2．リード・コントロール信号のアクティブ化

3．データ・トランシーバのイネーブルと方向制御の設定

メモリまたはI／O素子が直接に多重化アドレス／データ・バスに接続されていれば，TAZRLパラメータは，リード・コントロールがアクティブとなり選択された素子がバスを駆動できるようになる前に，8086がバスをフロート状態にすることを保証している．バス・サイクルの終わりで，次のバス・サイクルに対してアドレスを駆動する8086との競合を避けるならば，TRHAVパラメータは選択された素子が満たすべきバス・フロート・ディレイを指定している．次のバス・サイクルは，T4に続くCLK期間あるいはいくつかのCLK期間の後に開始される．

CPUにおいてリード・アクティブからデータが有効になるまでの最小ディレイは，2TCLCL - $TCLRL_{max}$ - TDVCL = 205ns である．最小パルス幅は，325ns の最小パルス幅を与える2TRLRH である．

$DT/\overline{R}$はバス・サイクルの初期に確定し，それ以上の考慮は必要としない．

読み込みの間，$\overline{DEN}$信号は，トランシーバが適当なデータ・セットアップ・タイムでCPUにデータを伝えるのを可能とし，必要なホールド時間の間はそれを続けなければならない．$\overline{DEN}$ターン・オン・ディレイは，8086によって要求されるデータが有効となる前に，TCLCL + $TCHCL_{min}$ − $TCVCTV_{max}$ − TDVCL = 129nsのトランシーバ・イネーブル・タイムを与える．

8086のデータ・ホールド・タイム$TCLDX_{min}$と最小の$\overline{DEN}$ターン・オフ・ディレイ$TCVCTX_{min}$は同じクロック・エッジに対して共に10nsの相対値を持つので，ホールド・タイムは保証される．さらに，次のバス・サイクルに対するアドレスで8086がバスを駆動する前に，$\overline{DEN}$はトランシーバをディスエーブルにしなければならない．最大の$\overline{DEN}$ターン・オフ・ディレイ（$TCVCTX_{max}$）を8086出力のアドレスに対する最小ディレイ（$TCLRH_{min}$）と比較すると，CPUが多重化バスをアドレスで駆動する少なくとも55ns前に，トランシーバはディスエーブルとなることが示される．

8.7.3　ライト・サイクルのタイミング

ライト・サイクルは次の3つの主要な機能から成る．

1．システムへのライト・データの供給

2．ライト・コマンドの発生

3．データ・バス・トランシーバのコントロール

ライト・データとライト・コマンドは共にT2の立上りエッジからイネーブルとなる．最小の$\overline{WR}$アクティブ・ディレイ$TCVCTV_{min}$と最大のライト・データ・ディレイTCLDVの比較から，ライトがアクティブになってから100ns後まではライト・データが有効ではないことが示される．したがって，システム中の素子は有効なデータを保証するために，ライト・コマンドの立上りエッジよりも立下りエッジでデータを捕えなければならない．HOLDまたは$\overline{RQ}/\overline{GT}$の入力によってバスから強制的に切り離されたときだけ，8086はライトの後にバスをフロート状態にし，そうでなければ，8086は単に次のバス・サイクルの開

始で出力の駆動をデータからアドレスに切り替える．リード・サイクルと同じく，次のバス・サイクルはT4に続くクロック期間あるいはいくつかのクロック期間の後に開始される．

$\overline{\text{WRITE}}$の立下りエッジの前，最小の2TCLCL $-$ TCLDV_{max} $+$ TCVCTX_{min} $=$ 300nsで8086からのデータは有効である．最小の$\overline{\text{WRITE}}$パルス幅はTWLWH＝340nsである．ライト後TWHDX，ＣＰＵは有効なデータを保持する．TWHDXの仕様は，TCLCH_{min}とTCHDX_{min}の関係から得られる結果を無効にする．これは，$\overline{\text{WR}}$後18nsだけライト・データが有効となることを示している．TCHDXの最小のバス・フロート・タイムは，TCVCTX＋TWHDX ＜TCLCH＋TCHDXのときだけ影響する．

トランシーバの方向のコントロール信号DT/ $\overline{\text{R}}$は，各リード・サイクルの終わりで伝送が条件づけられ，ライト・サイクルの間は変化しない．これにより，多重化バス上のアドレスを壊すことなく，アドレスが有効な間の，サイクルの初期においてトランシーバ・イネーブル信号$\overline{\text{DEN}}$をアクティブとすることができる．選択された素子に対するデータ・ホールド・タイムを保証するために，ライト後に最小のTCLCH_{min} $+$ TCVCTX_{min} $-$ TCVCTX_{max} $=$ 18nsで$\overline{\text{DEN}}$はディスエーブルとなる．再び最小のTCVTCXを最大のTCVCTXで評価したので，トランシーバのディスエーブルに対するライトの終わりからの実際のディレイは約60nsとなる．

8.7.4　インタラプト・アクノリッジのタイミング

インタラプト・アクノリッジ・シーケンスは，2つのインタラプト・アクノリッジ・バス・サイクルから成る．各サイクルのタイミングは，コントロール信号のタイミングとアドレス／データ・バスのタイミングを除いて，リード・サイクルのタイミングと同一である．

$\overline{\text{INTA}}$コントロール信号は，$\overline{\text{WR}}$コントロール信号と同じタイミングを有する．8086においてコントロールからデータ有効までの260nsのアクセス・タイムを与えるために，$\overline{\text{INTA}}$はT2の開始の110ns以内はアクティブである．8086のデータ・ホールド・タイムを満たすために，最小のTCVCTX_{min} $=$ 10nsに対してT4の立上りエッジに続いて$\overline{\text{INTA}}$コントロールがアクティブとなる．これは最小の$\overline{\text{INTA}}$パルス幅が300nsであることを保証するが，信号ディレイ・トラッキング（TCVCTX_{max} $=$ 110のとき，TCVCTX_{min} $=$ 50）を考慮に入れると，340nsの最小パルス幅となる．$\overline{\text{INTA}}$の最大インアクティブ・ディレイはTCVCTX_{max} $=$110nsで，8086は次のクロック・サイクルに対して15ns（TCLAV_{min}）まではバスを駆動しないので，出力をフロート状態にするためにローカル・バス上のインタラプト素子は105nsを利用できる．データ・バスにバッファが用いられていれば，出力をフロート状態にするために$\overline{\text{DEN}}$はローカル・バス・トランシーバに同じ時間を与える．

TCLAZ以内に $\overline{\text{INTA}}$サイクルの開始のT1から，多重化アドレス／データ・バスはフロート状態になる．多重化アドレス／ステータスの上位4つのラインはフロートとはならない．A19-A16に関するアドレス値は不定であるが，ステータス情報は有効である（S3＝０，S4＝０，S5＝IF，S6＝０，S7＝$\overline{\text{BHE}}$＝０）．多重化アドレス／データ・ラインは，$\overline{\text{INTA}}$バス・サイクルのT4に続くクロック期間までフロート状態のままである．このシーケンスは，2つの$\overline{\text{INTA}}$バス・サイクルに対して起きる．第2の$\overline{\text{INTA}}$バス・サ

イクルで8086によってリードされるインタラプト・ナンバーは，リード・サイクルのデータ・セットアップとホールドの時間を満たさなければならない．

$\overline{\mathrm{DEN}}$とDT/$\overline{\mathrm{R}}$の信号は，各$\overline{\mathrm{INTA}}$サイクルに対してイネーブルとなり，2つのサイクルの間でアクティブのままではない．これら2つの信号に対するタイミングは，$\overline{\mathrm{INTA}}$とリード・バス・サイクルにおいて同一である．

8.7.5 レディのタイミング

8086READY信号と8284に入力されるシステム・レディ信号（RDY）の詳細なタイミングの必要条件は，この章の初めに示してある．RDYは一般に，選択された素子のアドレス・デコードあるいはアドレスとコントロール信号$\overline{\mathrm{RD}}$，$\overline{\mathrm{WR}}$，$\overline{\mathrm{INTA}}$から生成される．

RDYがアドレス・デコードによってイネーブルになると，考慮すべき2つの場合が存在する．通常はノット・レディのシステムに対して，有効アドレスからレディを生成してウエート状態を挿入しないための時間は，$2TCLCL - TCLAV_{max} - TRIVCL_{max} = 255\mathrm{ns}$である．この時間は，バッファのディレイと，選択された素子がウエート状態を必要としないでRDYラインをハイに駆動するかどうかを判断するためのアドレス・デコードに利用できる．

ウエート・クロック期間が必要ならば，ユーザのハードウエアは適当なレディ・ディレイを備えていなければならない．アドレスは次のALEまで変化しないので，このバス・サイクル中，RDYは有効のままである．通常はレディのシステムでは，ウエート状態を必要とする選択された素子はまた，RDYラインをディスエーブルとするために255nsを有している．ユーザのハードウエアは，適当な数のウエート状態のクロック期間だけRDYを再びイネーブルとすることを遅らせなければならない．

RDYが$\overline{\mathrm{RD}}$コントロールによってイネーブルとなるならば，$TCLCL - TCLRL_{max} - TRIVCL_{max} = 15\mathrm{ns}$は外部ロジックで利用可能である．$\overline{\mathrm{WR}}$コントロールが用いられていれば，$TCLCL - TCVCTV_{max} - TRIVCL_{max} = 55\mathrm{ns}$が利用可能である．

アドレスあるいはコントロール信号で生成されるRDYの比較から，アドレス・デコーディングが最も良いタイミングを与える．システムが通常はレディでなければ，ウエート状態を必要としない素子に対してRDYを供給するためには，単にアドレス・デコーディングが用いられる．一方，ウエート状態を必要とする素子は，ウエート状態を発生させるために，アドレス・デコードとコントロール信号の組合せを用いる．システムが通常はレディならば，ウエート状態を必要としない素子はRDYに対して何も行なわない．一方，ウエート状態が必要な素子は，アドレス・デコードによってRDYをディスエーブルとして，RDYが再びイネーブルとなるまで遅らせるためにアドレス・デコードとコントロール信号を用いなければならない．システムがメモリに対してウエート状態を必要とせず，すべてのI/O素子に対する$\overline{\mathrm{RD}}$と$\overline{\mathrm{WR}}$に固定した数のウエート状態が必要ならば，M/$\overline{\mathrm{IO}}$信号は，ウエート状態のクロック期間が必要であることを前以って示すものとして用いることができる．これにより，アドレス・デコードをフィードバックさせることなしに，共通の回路でシステム全体のレディのタイミングをコントロールすることができる．

8.7.6　バス・コントロール移動のタイミング

詳細なHOLD/HLDAのタイミングについてはこの章の後で言及している．

$\overline{TEST}$入力は，WAIT命令の実行中にだけ，8086によってサンプルされる．検出を確実にするために，$\overline{TEST}$信号はWAIT命令の間，最小6クロックの期間はアクティブでなければならない．

8.8　マキシマム・モード・バスのタイミング

マキシマム・モードの8086バス動作は，論理的にミニマム・モードと同等である．詳細なタイミングの解析には，8086 CPUと8288バス・コントローラによって生成される信号が含まれる．

ミニマム・モードの8086によって与えられる供給信号に加えて，8288はシステムの融通性を拡張する付加的なコントロール信号を備えている．以下の解説で，信号の関係を検討すれば，同等のミニマム・モードの有用性よりも，適切なマキシマム・モードの有用性を用いることは確実となる．

8.8.1　アドレスとALE

マキシマム・モードにおいて，アドレス情報は常に8086から得られるが，ALEストローブは8288バス・コントローラによって作られる．ALEと有効アドレスとの間のワースト・ケースの関係を決定するためには，8086からのステータス$\overline{S0}$－$\overline{S2}$に関して8288のALEのアクティブについて解析する必要がある．

マキシマム・モードのタイミング図は，ALE生成の2つの可能性のあるディレイの経路を示す．第1はTCHSV＋TSVLHで，これはT1に先行するクロック期間の立上りエッジから測定される．第2の経路はTCLLHで，これはT1の開始から測定される．8288はステータス・ラインをパッシブ状態（$\overline{S0}$，$\overline{S1}$，$\overline{S2}$＝1，1，1）にすることからバス・サイクルを初期化するので，クロックがハイの時間が最小（$TCHCL_{min}$）の間に8086のステータスの発行（$TCHSV_{max}$）が遅れると，T1の開始までにステータスの変化は起きず，ALEはステータスが変化した後TSVLHに発行される．T1の開始より前にステータスが変化すると，8288はT1の開始後TCLLHまでALEを発行しない．結果としてワースト・ケースのALEをイネーブルにするディレイ（T1の開始に関して）は，$TCHSV_{max}+TSVLH_{max}-TCHCL_{min}=58ns$である．

ALEをイネーブルとするディレイを無視すれば，ALEの立下りエッジは8288においてT1の正のクロック・エッジでトリガされる．結果的に最小のALEパルス幅は，TCHLL＝0を仮定すれば$TCLCH_{max}-58ns=75ns$である．$TCHCL_{min}$は58nsのALEイネーブル・ディレイを導びくと仮定されているので，$TCLCH_{max}$を用いる必要がある．8288あるいは8283においてアドレスを捕えるために，ALEの立下りエッジの前$TCLCH_{min}+TCHLL_{min}-TCLAV_{max}=8ns$はアドレスが有効であることが保証される．ここでは再び安全をみてTCHLL＝0を仮定した．アドレスとALEは別の素子で駆動されるので，

A.C.特性のトラッキングは仮定できないことに注意.

ラッチに対するアドレス・ホールド・タイムはT1の終わりまでアドレスが有効であることによって保証され，一方ALEはT1において正のクロック遷移から最大15nsでディスエーブルになる（$TCHCL_{min} - TCHLL_{max} = 52ns$のアドレス・ホールド・タイム）.アドレスからステータスとライト・データあるいはトライステート（リードに対して）への多重化バスの遷移は，ミニマム・モードのタイミングと同一である．また，アドレス有効のディレイ（TCLAV）は有効アドレス確定におけるクリティカルな経路のままなので，有効なデータとレディに対するアドレス・アクセス・タイムはミニマム・モード・システムと同じである.

8.8.2 リード・サイクルのタイミング

マキシマム・モード・システムは，8086と8288によって別々に発生される2つのリード信号を提供する.8086の$\overline{RD}$出力信号のタイミングはミニマム・モード・システムと同一であるが，8288によって発生させられるリード・コントロール信号のA.C.特性は非常に良い．したがって分離されてバッファを持つシステム・バス上の素子は，8288のリード・コントロール信号を用いるべきである.8086の$\overline{RD}$信号は，多重化バス上に直接位置する素子に用いられる.

以下の評価では，8288のリード・コントロール信号のタイミングについてだけ考慮する.

8288は，別々のメモリとI/Oのリード・コントロール信号（$\overline{IORC}$と$\overline{MRDC}$）を出力し，この2つは同じA.C.特性を持つ．このコントロール信号は，T2の開始後TCLMLに発行され，T4の開始後TCLMHに終結する．最小のコントロール・パルスの長さは$2TCLCL - TCLML_{max} + TCLML_{min} = 375ns$である.8086における有効データのアクセス・タイムは$2TCLCL - TCLML_{max} - TDVCL_{max} = 335ns$となる.8288はバッファを用いたデータ・バスを有するシステムのために設計されているので，コントロール信号$\overline{IORC}$と$\overline{MRDC}$は8086が多重化バスをフロート状態にする前にイネーブルとなる．したがって，コントロール信号$\overline{IORC}$と$\overline{MRDC}$は多重化バスに直接接続している素子で用いられてはならず，さもなければ結果として，8086のバス・フロートと素子のターン・オンの間にバス競合が生じる.

データ・バス・トランシーバの方向制御はT1で確定される．トランシーバは$\overline{DEN}$によってT2の正のクロック遷移までイネーブルとなる．これにより，$TCLCH + TCVNV_{min} = 123ns$の8086バス・フロート・ディレイを与え，$TCHCL_{min} + TCVNV_{max} - TDVCL_{max} = 187ns$の間，8086において有効なデータに対してトランシーバはアクティブとなる.$\overline{DEN}$とコントロールの信号は共に，T4に対して最小10nsは有効なので，8086のデータ・ホールド・タイムTCLDZは保証される.$\overline{DEN}$ディスエーブルの最大45ns（$TCVNX_{max}$）は，次の8086のバス・サイクルの開始（同じクロック・エッジから最小215ns）までにトランシーバがディスエーブルとなることを保証している.T4の正のクロック遷移において，次のバス・サイクルでの可能性のある書き込み動作の準備のために，DT/$\overline{R}$は送信に戻る．システムのメモリとI/Oの素子はバッファのあるシステム・バス上に存在するの

で，次のバス・サイクルに対する素子が選択される（およそ2TCLCL）前かあるいはトランシーバがバス上にライト・データを駆動する（およそ2TCLCL）前に，その出力をフロート状態にしなければならない．

8.8.3　ライト・サイクルのタイミング

マキシマム・モードでは，8288はメモリとI/Oに対して標準のものと先行したコントロール信号（$\overline{\mathrm{MWTC}}$，$\overline{\mathrm{AMWC}}$，$\overline{\mathrm{IOWC}}$，$\overline{\mathrm{AIOWC}}$）を供給する．アドバンスド・ライト・コントロール信号は，標準ライト・コントロール信号の前の1クロック期間を通してアクティブである．アドバンスド・ライト・コントロール信号のタイミングはリード・コントロール信号のタイミングと同一である．

アドバンスド・ライト・パルス幅は$2TCLCL - TCLML_{max} + TCLMH_{min} = 375ns$であり，一方，標準ライト・パルス幅は$TCLCL - TCLML_{max} + TCLMH_{min} = 175ns$である．選択された素子に対するライト・データのセットアップ・タイムは，8086からのデータが有効となるディレイ（TCLDV），またはトランシーバ・イネーブル・ディレイ(TCVNV）の関数となる．有効なライト・データに対するワースト・ケースのディレイは，$TCLDV = 110ns$からトランシーバの伝播ディレイを引いたものになる．これは，データがアドバンスド・ライト・コントロール信号の立上りエッジ後100nsまで有効ではないが，標準ライト・コントロール信号の立上りエッジの前およそ$TCLCL - TCLDV_{max} + TCLML_{min} = 100ns$は有効であることを意味する．どちらのライト・コントロール信号の立下りエッジ前$2TCLCL - TCLDV_{max} + TCLMH_{min} = 300ns$も，データは有効である．

アドバンスド・ライト・コントロールに対するデータとコントロール信号のオーバラップは300nsであり，一方標準ライト・コントロールとのオーバラップは175nsである．ライト・コントロール後，最小$TCLCH_{min} - TCLMH_{max} + TCVNX_{min} = 85ns$でトランシーバはディスエーブルとなり，8086は最小$TCLCH_{min} - TCLMH_{max} + TCHDZ_{min} = 85ns$の間は有効なデータを供給する．これはライト・コントロール後に85nsのライト・データ・ホールドを保証する．トランシーバは，後続のリード・バス・サイクルのためにその方向を変える前，$TCLCL - TCVNX_{max} + TCHDTL_{min} = 155ns$（$TCHDTL = 0$と仮定）でディスエーブルとなる．

8.8.4　インタラプト・アクノリッジのタイミング

マキシマム・モードの$\overline{\mathrm{INTA}}$シーケンスは論理的にミニマム・モードのシーケンスと同一である．両者のインタラプト・アクノリッジ・サイクルのトランシーバ・コントロール（$\overline{\mathrm{DEN}}$とDT/$\overline{\mathrm{R}}$）と$\overline{\mathrm{INTA}}$コントロールのタイミングは，リード・サイクルのトランシーバ・コントロールのタイミングと同一である．ミニマム・モード・システムと同じく，各$\overline{\mathrm{INTA}}$バス・サイクルに対してT1の立上りエッジから多重化アドレス／データ・バスはフロート状態となり，各$\overline{\mathrm{INTA}}$サイクルのT4の後まで8086によって駆動されない．

第2の$\overline{\mathrm{INTA}}$サイクルの間に外部ハードウエアによって返されるベクタのセットアップとホールドのタイムは，リード・バス・サイクルに対するデータのセットアップとホール

ドと同じである．インタラプト・ベクタを与える素子がローカル・バスに接続されていれば，8086 のバス・フロートから $\overline{\mathrm{INTA}}$ のコマンドがアクティブになるまで $\mathrm{TCLCL} - \mathrm{TCLAZ}_{max} + \mathrm{TCLML}_{min} = 130\mathrm{ns}$ が利用できる．ローカル・バス上の選択された素子は，ＤＥＮがなお 8288 によって生成されているので，システム・データ・バスのトランシーバをディスエーブルとしなければならない．

8288 がＩＯＢ（I/O バス）モードでなければ，8288 の MCE/$\overline{\mathrm{PDEN}}$ 出力は MCE 出力となる．この出力は $\overline{\mathrm{INTA}}$ サイクルの間はアクティブで，T1 の間のＡＬＥ信号とオーバラップしている．ＭＣＥはマスタ8259Ａからのカスケード・アドレスを上位 AD15-AD8 ラインの３つに送出するゲートとして利用できる．またＭＣＥによってＡＬＥがカスケード・アドレスをアドレス・ラッチにラッチすることが可能となる．

次にアドレス・ラインは，システム・バス上に位置するスレーブ8259Ａに対してＣＡＳアドレス選択を供給するために用いられる（この方法の記述については図8-32を参照）．ＭＣＥは，各 $\overline{\mathrm{INTA}}$ サイクルに対してステータスあるいはＴ1開始の 15ns 以内はアクティブである．最初の $\overline{\mathrm{INTA}}$ サイクルで80ns まで8086がバスをフロート状態にすることは保証していないので，最初のサイクルの間にＭＣＥは多重化バスでのＣＡＳラインをイネーブルにはしない．最初のＭＣＥは，ＭＣＥと $\overline{\mathrm{LOCK}}$ のゲートによって禁止できる．8086の $\overline{\mathrm{LOCK}}$ 出力は最初の $\overline{\mathrm{INTA}}$ サイクルのＴ2の間にアクティブとなり，第２の $\overline{\mathrm{INTA}}$ サイクルのＴ2の間にディスエーブルとなる．ＭＣＥとの $\overline{\mathrm{LOCK}}$ のオーバーラップにより，最初のＭＣＥをマスクして，第２のＭＣＥをカスケード・アドレスのローカル・バスへのゲートとすることができる．

8259Ａは第２の $\overline{\mathrm{INTA}}$ バス・サイクルまでカスケード・アドレスを供給しないので，情報は失なわれない．ＡＬＥと同じく，ＡＬＥの立下りエッジで 75ns のＣＡＳアドレス・セットアップを可能とするために，ＭＣＥは T1 の開始の58ns 以内で有効であることが保証される．ラッチに対してデータ・ホールド・タイムを与えるために，ＡＬＥ後 $\mathrm{TCHCL}_{min} - \mathrm{TCHLL}_{max} + \mathrm{TCLMCL}_{min} = 52\mathrm{ns}$ の間，ＭＣＥはアクティブである．

8288 がＩＯＢモードに切り替えられていれば，ＭＣＥ出力は $\overline{\mathrm{PDEN}}$ となり，すべてのI/O 参照は，分離システム・バス上ではなくローカル・バス上の素子として仮定される．$\overline{\mathrm{INTA}}$ サイクルは I/O サイクルと考えられるので，すべての割り込みはローカル・システム・バスからのものと仮定され，カスケード・アドレスはシステム・アドレス・バス上には送出されない．さらに，システム・バス上では I/O の伝送が起きないので，$\overline{\mathrm{DEN}}$ 信号はイネーブルとならない．ローカル I/O バスもまたトランシーバによるバッファを有するならば，このトランシーバをイネーブルとするために $\overline{\mathrm{PDEN}}$ 信号が用いられる．$\overline{\mathrm{PDEN}}$ のA.C.特性はＤＥＮと同一であり，$\overline{\mathrm{PDEN}}$ は I/O の参照でイネーブルとなり，ＤＥＮは命令またはデータのメモリ参照でイネーブルとなる．各種のモードのシステムの意味は後の章で述べる．

8.8.5 レディのタイミング

アドレス有効のタイミングに基づいては，レディのタイミングはマキシマム・モードと

ミニマム・モードのシステムで同じである．8284 での 8288 のコントロール有効からＲＤＹの有効までのディレイは，$TCLCL - TCLML_{max} - TRIVCL_{min} = 130ns$ である．この時間は，ウエート状態のクロック期間が挿入される必要性を判断するために外部回路で用いられる．外部回路は，ウエート状態を挿入するためにＲＤＹをディスエーブルにするか，ウエート状態を避けるためにＲＤＹをイネーブルにしなければならない．$\overline{INTA}$, すべてのリード・コントロール，そしてアドバンスド・ライト・コントロールはこのタイミングを与える．

標準のライト・コントロールは，ＲＤＹが有効となる後まで有効とはならない．標準のものとアドバンスド・ライト・コントロールは，すべてのライト・バス・サイクルに対して 8288 によって発生させられるので，選択された素子が標準のライト・コントロールを用いても，ＲＤＹを示すためにアドバンスド・ライト・コントロールが用いられる．

8.8.6　その他の考察

$\overline{RQ}/\overline{GT}$ のタイミングについては，この章の後で言及する．

マキシマム・モードで考慮すべき唯一の信号はキュー・ステータス・ラインＱＳ０とＱＳ１である．この信号は，アイドルとウエートのクロック期間を含んで，各クロック期間の立上りエッジ（ハイからローへの遷移）で変化する．キュー・ステータスは，ＢＩＵの動作とは独立に，エグゼキューション・ユニットのステータスを表わす．外部ロジックは，クロック・パルスのローからハイの遷移でこのラインをサンプルする．サンプル時は，ＱＳ０とＱＳ１の信号は以前のクロック期間におけるキュー動作を識別しており，したがってＣＰＵの動作とは１クロック期間だけのずれがある．

$\overline{TEST}$ 入力条件は，ミニマム・モードに対して述べたものと同一である．

8.9　バス・コントロールの移動（HOLD/HLDA と $\overline{RQ}/\overline{GT}$）

8086 自身とバス・マスタとして動作可能な他の素子の間で，ローカル・バスのコントロールを移すために用いられるプロトコル信号を 8086 は有している．ミニマム・モード構成は，8080Ａと 8085 のシステムと同一のシングル・レベルのハンドシェイクを与える．マキシマム・モード構成は，２つのレベルの優先権でシステム構成を２つのレベルの交互のバス・マスタに拡張して，ＣＰＵのピンをより有効に用いるエンハンスト・パルス・シーケンス・プロトコルを有している．このプロトコル信号は 8086 ローカル・バスのコントロールを判定する．これをシステム・バスの判定と混同してはならない．

8.9.1　ミニマム・モード

ミニマム・モード 8086 システムでは，ＣＰＵに対するホールド・リクエスト入力（HOLD）とＣＰＵからのホールド・アクノリッジ出力（HLDA）を用いる．ローカル・バスのコントロールを得るために，素子はＣＰＵに対して HOLD を主張して，バスを駆動する前

にHLDAを待たなければならない．8086がバスを放棄できれば，$\overline{\mathrm{RD}}$，$\overline{\mathrm{WR}}$，$\overline{\mathrm{INTA}}$とM/$\overline{\mathrm{IO}}$のコントロール・ライン，$\overline{\mathrm{DEN}}$とDT/$\overline{\mathrm{R}}$バス・コントロール・ライン，そして多重化アドレス/データ/ステータス・ラインをフロート状態にする．ALE信号はフロートとはならない．CPUはHLDAでローカル・バスに対する要求の受付けを通知する．これにより，要求している素子がローカル・バスのコントロールを得ることができる．

要求素子は，もうそれ以上ローカル・バスを必要としなくなるまで，HOLDリクエストをアクティブに維持しなければならない．8086に対するHOLDリクエストは，直接にバス・インターフェイス・ユニットに影響を与え，間接的にエグゼキューション・ユニットに影響する．

エグゼキューション・ユニットは，さらに命令が必要となるかオペランドの転送が必要となるまで内部キューから実行を続ける．これにより，CPUと補助バス・マスタの動作の間に程度の小さいオーバラップが可能となる．要求元のマスタがHOLD信号を落とすと，8086はHLDAを落として応答する．8086はバスとコントロールの信号を再び駆動しない．この信号は，8086がバス転送を行なう必要があるまでフロート状態を続ける．HOLDが落ちたときに8086は，なおその内部キューから実行しているので，どの素子もバスを駆動していない期間が存在する．

バス・コントロールの遷移の間にコントロール・ラインが最小のVIHレベル以下にドリフトするのを防ぐために，22キロオームのプルアップ抵抗をバス・コントロール・ラインに接続する必要がある．**図8-35**のタイミング図は，CPUのクロックに関して，HOLDのサンプル，バスのフロート，HLDAのイネーブル/ディスエーブルのタイミングの8086におけるバス・コントロール・ハンドシェイク・シーケンスを示している．

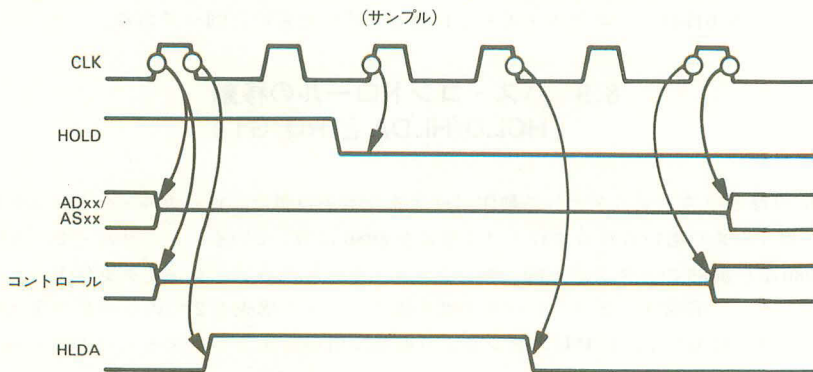



図8-35　HOLD/HLDAシーケンス

確実なシステム動作を確保するために，設計者は，要求元素子が8086のコントロール放棄の前にバスのコントロールを主張しないように，また，この素子が8086のバス駆動の前にバスのコントロールを放棄するように，保証しなければならない．HLDAと8086がバスをフロート状態とする間の最大ディレイは，$\mathrm{TCHDZ}_{max} - \mathrm{TCHCL}_{min} - \mathrm{TCLHAV}_{min} = 10\mathrm{ns}$である．システムが10nsのオーバラップを許すことができなければ，8086からの

HLDAのアクティブは素子に対してディレイを持たなければならない．

8086がローカル・バス上のコントロール信号を駆動するまでのHOLDのインアクティブからの最小ディレイは，$THVCH_{min}$ + 3TCLCL = 635nsであり，多重化バスの駆動のためには，このディレイは$THVCH_{min}$ + 3TCLCL + TCHCL = 701nsとなる．素子が指定された時間内にローカル・バスを解放しなければ，8086に対するHOLDのインアクティブはディレイを持たなければならない．HLDAのインアクティブからバスの駆動までのディレイは，ローカル・バスのコントロール信号に対してTCLCL + $TCLCH_{min}$ − $TCLHAV_{max}$ = 158ns，そしてローカル・データ・バスに対してTCLCL − $TCLHAV_{max}$ = 240nsである．

(1) HOLDに対するHLDAの潜伏

HOLDリクエストに対する応答の決定は，バス・インターフェイス・ユニットによって行なわれる．この決定に影響する主要な要因には，現在のバス動作，CPU内部の$\overline{LOCK}$信号の状態（ソフトウエアのLOCKプレフィックスによってアクティブになる），そして保留状態のインタラプトがある．

$\overline{LOCK}$がアクティブでなく，インタラプト・アクノリッジ・サイクルが進行中でなく，そしてHOLDリクエストが受信されたときにBIU（バス・インターフェイス・ユニット）がT4あるいはTIのクロック期間を実行していれば，HLDAに対する最小の潜伏期間は次のようになる．

35 ns	THVCH min（ホールド・セットアップ）
65 ns	TCHCL min
200 ns	TCLCL（バス・フロート・ディレイ）
10 ns	TCLHAV min（HLDAディレイ）
———————	
310 ns	@ 5 MHz

上記条件でのHLDAに対する最大の潜伏期間は次のようになる．

34 ns	（セットアップ・タイム）
200 ns	次のサンプルに対するディレイ
82 ns	TCHCL max
200 ns	TCLCL（バス・フロート・ディレイ）
160 ns	TCLHAV max（HLDA）
———————	
677 ns	@ 5 MHz

ホールド・リクエストが受信されたときに，ちょうどCPUがバス・サイクルの初期化を行なっていれば，ワースト・ケースの応答時間は次のようになる．

34 ns	THVCH
82 ns	TCHCL max
7*200	バス・サイクル実行
N*200	N個のウエート状態のバス・サイクル
160 ns	TCLHAV max（HLDAディレイ）
———————	
1.676 μs	@ 5 MHz，ウエート状態なし

ホールド・リクエストのミスに対する200nsはバス・サイクル実行のディレイに含まれていることに注意．オペランドの転送が奇数バイト境界に対するワード転送ならば，この転送のために2つのバス・サイクルが実行される．ＢＩＵはこの2つのバス・サイクルの間のホールド・リクエストを受け付けない．このタイプの転送は，前記最大潜伏期間をさらに4つのクロック期間とN個のウエート状態だけ延長する．バス・サイクルにウエート状態がなければ，最大値は2.476マイクロ秒となる．

ミニマム・モードの8086にはハードウエアの$\overline{\text{LOCK}}$出力はないが，それでも命令の中にはソフトウエアのLOCKプレフィックスが含まれる．ＣＰＵは内部的に，マキシマム・モードの8086と同様にLOCKプレフィックスに反応する．したがってLOCKは，プレフィックスに続く命令が完了するまでホールド・リクエストを受け付けられない．その結果，ＣＸを（ＢＸ）に加えてその結果を（ＢＸ）にストアするADD（ＢＸ），ＣＸ などの，１つ以上のメモリ参照を行なう命令は，メモリ参照の間に他のバス・マスタがバスのコントロールを獲得することなしに実行できる．$\overline{\text{LOCK}}$信号は命令実行より１クロック期間多い間アクティブなので，HLDAに対する最大潜伏時間は次のようになる．

34 ns	THVCH
200 ns	次のサンプルに対するディレイ
82 ns	TCHCL max
(M＋1)*200 ns	LOCK 命令実行
200 ns	セットアップ HLDA（内部）
160 ns	TCLHAV max（HLDA ディレイ）
――――――――――	
(M*200 ns)＋876 ns	@ 5 MHz
	Mはロックされた命令実行のクロック数

ホールド・リクエストがインタラプト・アクノリッジ・シーケンスの開始に行なわれると，HLDAに対する最大潜伏時間は次のようになる．

34 ns	THVCH
82 ns	TCHCL max
2600 ns	INTAの13クロック・サイクル
160 ns	TCLHAV max
―――――――	
2.876 μs	@ 5 MHz

(2) ミニマム・モードのDMA構成

ミニマム・モードのHOLD/HLDA信号の代表的な利用は，インテルの8257-5あるいは8237ＤＭＡコントローラなどの，ＤＭＡコントローラ素子とのバス・コントロールの交換である．**図8-36**は，8257-5を用いたこのタイプの構成を機能的に示している．

ＤＭＡコントローラは8086のローカル多重化アドレス・データ・バスの上位に位置し，8086とA15-A8の分離アドレス・ラッチを共有している．8257-5のレジスタは，バスの上位を通してアクセスしなければならない．したがって，奇数アドレスのレジスタ（Ａ０＝１）は奇数I/Oアドレスに対するバイト転送としてアクセスされ，偶数アドレスのレジスタはデータが上位バイトで転送されるものとしてワードI/Oによってアクセスされる．80

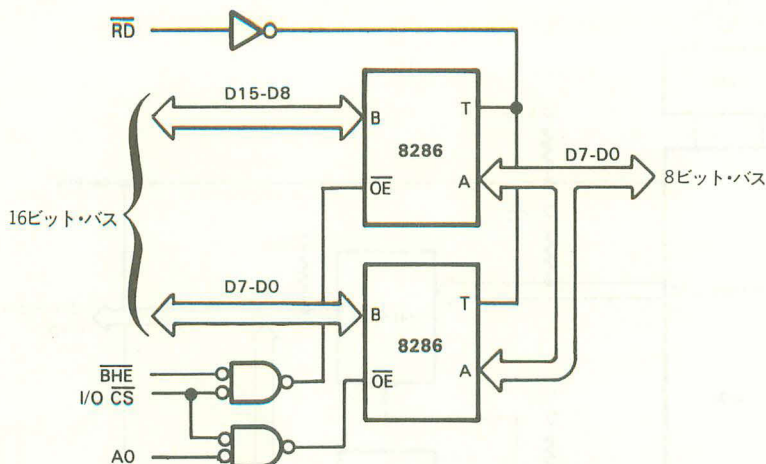
図8-36　ミニマム・モードを用いたDMA

86の読込みと書込みのコントロール信号は，8257-5 の必要条件に適合する別々のI/O とメモリのコントロールを得るために，分離しなければならない．8257-5 からのＡＥＮコントロールで，8086 のコントロール信号と，下位（A7-A0）と上位（A19-A16）のアドレス・バスのラッチをディスエーブルとする必要がある．また，ＡＥＮはA15-A8 のアドレス・ラッチに対する 8257-5 のアドレス・ストローブ（ADSTB）を選択しなければならない．データ・バスにバッファが用いられていれば，$\overline{\text{DEN}}$ ラインのプルアップ抵抗はバッファをディスエーブルに保つ．ＤＭＡコントローラは，メモリとI/O の間でバイトを転送するだけである．ＤＭＡコントローラは，以下に示されている 16 ビットから 8 ビットへのバス多重化回路から得られる 8 ビット・バス上にI/O 素子が存在していることを必要とする．アドレス・ライン A7-A0 は直接 8257 によって駆動され，$\overline{\text{BHE}}$ は A0 を反転することによって生成される．A19-A16 が用いられていれば，これは固定値あるいはソフトウエアで初期化される値を持つ付加的なポートによって与えられなければならない．この付加的ポートは，ＡＥＮによってアドレスがイネーブルとなる必要がある．

図8-37 は，システム・バスに接続されている 8257 を示す．

8257-5 からの上位アドレスの保持に別のラッチを用いて，示されているように出力をアドレス・バスに接続すれば，16 ビットのＤＭＡ転送が得られる．この構成において，ＡＥＮはワード転送を可能とするために A0 と $\overline{\text{BHE}}$ を同時にイネーブルにする．それでもなお，ＡＥＮはコントロールとアドレスのバスに対するＣＰＵのインターフェイスをディスエーブルとしている．

8.9.2 マキシマム・モード（$\overline{\text{RQ}}/\overline{\text{GT}}$）

マキシマム・モード 8086 構成は，大きく異なるバス・コントロール移動のプロトコルをサポートしている．

(1) 共有システム・バス（$\overline{\text{RQ}}/\overline{\text{GT}}$ の選択）

マキシマム・モードの $\overline{\text{RQ}}/\overline{\text{GT}}$ シーケンスは，8086 と，全体的にローカル・バス上に位

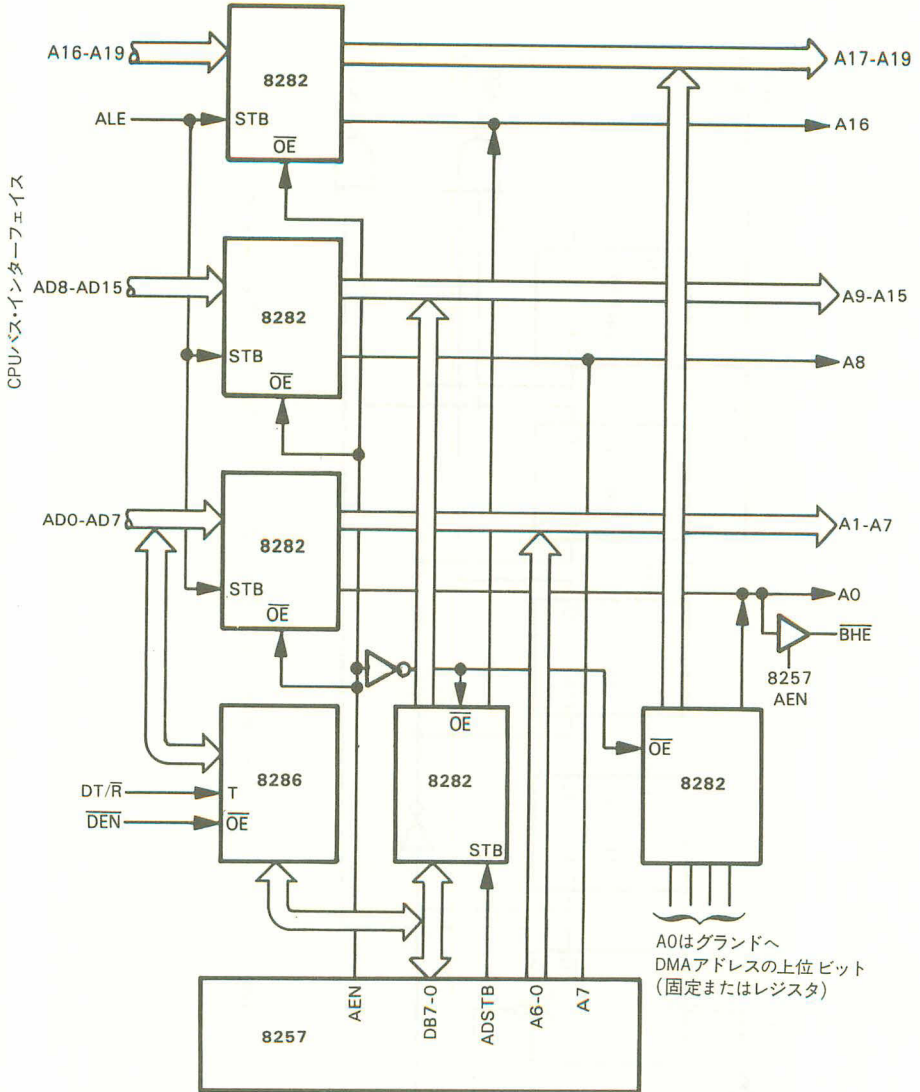



図8-37　8086ミニマム・モード・システムの16ビット・データ転送のシステム・バスにおける8257

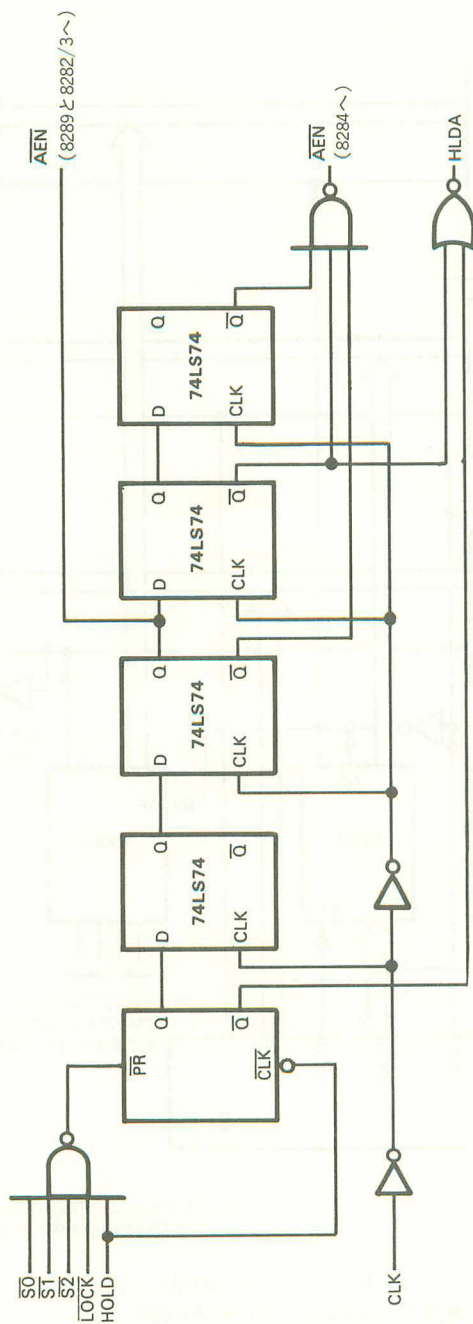



図8-38　マキシマム・モード8086のHOLDから$\overline{AEN}$のディスエーブル信号への変換

置してシステム・バスに対する完全なCPUのインターフェイスを共有するDMAコントローラなどのような別のバス・マスタとの間で，ローカル・バスのコントロールを移すために用いられる．システム・バスに対する完全なCPUインターフェイスには，アドレス・ラッチ，データ・トランシーバ，8288 バス・コントローラ，そして 8289 マルチマスタ・バス・アービタが含まれる．システムの別のバス・マスタが直接 8086 ローカル・バス上に位置していなければ，システム・バスの判定は必要であり，ローカル・バスの判定は必要でない．マルチマスタ・バス判定には 8289 バス・アービタが必要であり，$\overline{\mathrm{RQ}}/\overline{\mathrm{GT}}$ ロジックは用いられない．

簡単なHOLD/HLDA プロトコルを有する素子が，ちょうど1つのCPUが接続されているシステム・バスのコントロールを得るならば，**図8-38**の回路が用いられる．

実際この回路は，他のバス・マスタがホールド・リクエストを発行したときに，システム・バスからCPUを分離する簡単なバス・アービタである．8086 がアイドル状態（$\overline{\mathrm{S0}}$，$\overline{\mathrm{S1}}$，$\overline{\mathrm{S2}}$＝1）を示し，$\overline{\mathrm{LOCK}}$ がインアクティブでホールド・リクエストがアクティブのとき，回路の出力$\overline{\mathrm{AEN}}$（アクセス・イネーブル）は 8288 と 8284 をディスエーブルにする．$\overline{\mathrm{AEN}}$がインアクティブになると，8288 はコントロール出力をフロート状態にしてDENをディスエーブルとし，これによってデータ・バスのトランシーバはフロート状態になる．$\overline{\mathrm{AEN}}$はまたアドレス・ラッチ（8282 または 8283）の出力をフロート状態にしなければならない．この動作でシステム・バスから 8086 が除去され，要求元素子のシステム・バスの駆動が可能となる．

8284 への$\overline{\mathrm{AEN}}$信号は READY 入力をディスエーブルとして，8086 がシステム・バスのコントロールを取戻すまで待つために，8086 によってバス・サイクルの初期化を行なわせる．CPUはこの期間にローカル・バスを能動的に駆動する．

要求元素子は，8086 開始のバス・サイクル，ロックされた命令の実行，あるいはインタラプト・アクノリッジ・サイクルの間，システム・バスのコントロールを得られない．8086 からの $\overline{\mathrm{LOCK}}$ 信号は，8086 のバス・コントロールの維持を保証するために，$\overline{\mathrm{INTA}}$ サイクルの間アクティブである．ミニマム・モード 8086 の HLDA 応答と異なり，奇数アドレス境界のワード・オペランド転送の連続したバス・サイクルの間に，要求元素子はバスのコントロールを得ることができる．要求元素子の特性に依存して，他の 74LS74 の出力の1つをその素子に対する HLDA を生成するために利用できる．これは，バスを用いる前にディレイを必要とする素子を接続するときに有用である．

システム・バスの動作が終了すると，そのバス・マスタはシステム・バスのコントロールを放棄して HOLD リクエストを落とさなければならない．$\overline{\mathrm{AEN}}$がアクティブとなった後，アドレス・ラッチとデータ・トランシーバはイネーブルとなるが，8086開始のバス・サイクルが保留状態ならば，最小 105ns のディレイまたは最大 275ns のディレイが経過するまで，8288 はコントロール・ラインを駆動しない．システムが通常はノット・レディならば，選択された素子が転送を終了すると 8086 に READY を返して，8284 の $\overline{\mathrm{AEN}}$ 入力は直ちにイネーブルになる．

システムが通常はレディならば，標準バス・サイクルに等価なアクセス時間を与えるの

に十分長く8284 $\overline{\mathrm{AEN}}$入力を遅らせなければならない.設計において74LS74ラッチは,HOLDを解放した後に別の素子がシステム・バスをフロートにするために最小 TCLCHを与える.また,8284のREADY検出を可能にするまえに,2TCLCLのアドレス・アクセスと 2TCLCL－TAEVCH_{max}（8288コマンド・イネーブル・ディレイ）のコントロール・アクセスも与える.HLDAが図8-38に示されているように生成されるならば,HLDAを発行する前に8086がバスを解放するためにTCLCLを利用でき,HLDAはHOLDを失なうとすぐに落とされる.

マキシマム・モードの8086にインターフェイスするためのこの技法を用いた8257-5の回路構成は,図8-37から導くことができる.8257-5はアドレス・ラインA15-A8のバッファのための自分自身のアドレス・ラッチを有する.8257-5はアドレス・バス上へのラッチをイネーブルにするためにそのAEN出力を用いる.

この回路でのHOLDからHLDAへの最大潜伏期間は,HOLDが発行されたときのシステムの状態に依存している.アイドルのシステムで,最大潜伏期間は,NANDゲートを通しての伝播ディレイとR/Sフリップフロップの（TD1）＋2TCLCL＋TCLCH_{max}＋74LS74と74LS02の伝播ディレイ（TD2）である.ロックされた命令では,TD1＋TD2＋(M＋2)＊TCLCL＋TCLCH_{max}となる.ただし,Mはロックされた命令の実行に必要なクロック数である.インタラプト・アクノリッジ・サイクルに対して,潜伏期間はTD1＋TD2＋9＊TCLCL＋TCLCH_{max}となる.

(2) 共有ローカル・バス（$\overline{\mathrm{RQ}}/\overline{\mathrm{GT}}$の使用法）

$\overline{\mathrm{RQ}}/\overline{\mathrm{GT}}$プロトコルは,1つまたは2つの他の命令セット拡張プロセッサ（コープロセッサ）あるいは特殊な機能のプロセッサを,8086のローカル・バスと直接接続することを可能とするために開発されたものである.8086の$\overline{\mathrm{RQ}}/\overline{\mathrm{GT}}$ピンは,バス・コントロール交換の完全なプロトコルをサポートする.

バス・コントロール交換シーケンスは,ローカル・バスのコントロールを得るための別のバス・マスタからのリクエスト,ローカル・バスが放棄されたことを示すための8086からの許可,そして終了時の別のバス・マスタからの解放パルスから構成される.2つの$\overline{\mathrm{RQ}}/\overline{\mathrm{GT}}$ピン（$\overline{\mathrm{RQ}}/\overline{\mathrm{GT0}}$と$\overline{\mathrm{RQ}}/\overline{\mathrm{GT1}}$）には優先順位があり,$\overline{\mathrm{RQ}}/\overline{\mathrm{GT0}}$は高い優先権を持つ.優先権は,どちらかに応答する前に両方のピンでリクエストを受けたときだけ意味を持つ.たとえば,$\overline{\mathrm{RQ}}/\overline{\mathrm{GT1}}$でリクエストを受け,次いで$\overline{\mathrm{RQ}}/\overline{\mathrm{GT1}}$の許可の前に$\overline{\mathrm{RQ}}/\overline{\mathrm{GT0}}$でリクエストを受けると,$\overline{\mathrm{RQ}}/\overline{\mathrm{GT0}}$は$\overline{\mathrm{RQ}}/\overline{\mathrm{GT1}}$に対して優先権を持つ.しかし,$\overline{\mathrm{RQ}}/\overline{\mathrm{GT1}}$がすでに優先権を認められていれば,$\overline{\mathrm{RQ}}/\overline{\mathrm{GT0}}$でのリクエストは$\overline{\mathrm{RQ}}/\overline{\mathrm{GT1}}$で解放パルスが受かるまで待つ必要がある.

バス・インターフェイス・ユニットでの要求／許可の相互作用シーケンスはHOLD/HLDAと類似している.8086は,命令をフェッチするためかオペランドの処理のためにバス・サイクルを要求するまで,その内部キューからの命令の実行を続ける.しかし,8086がバスを必要とする前に解放のパルスを受け取れば,バス・サイクルが必要となるまでバスを駆動しない.

リクエスト・パルスを受け取ると,8086は多重化アドレス／データ・バス,$\overline{\mathrm{S0}}$,$\overline{\mathrm{S1}}$,

$\overline{\text{S2}}$のステータス・ライン，$\overline{\text{LOCK}}$と$\overline{\text{RD}}$をフロートにする．この動作では，8288のコントロール出力をディスエーブルとせず，またアドレス・ラッチをディスエーブルともせず，アドレス・バスを駆動し続ける．8086の出力がフロートの間にパッシブ・ステートを維持するために，8288は$\overline{\text{S0}}$，$\overline{\text{S1}}$，$\overline{\text{S2}}$のステータス・ラインに内部プルアップ抵抗を含む．このパッシブ・ステートは，データ・バスのバッファのトランシーバをイネーブルとするための8288のコントロール出力の開始，あるいはDENのアクティブ化を防止する．$\overline{\text{RQ}}$発行の素子が8288を用いないならば，8288$\overline{\text{AEN}}$入力をディスエーブルにすることによって8288のコントロール出力をディスエーブルにしなければならない．また，要求元素子によって用いられていないアドレス・ラッチは，要求元素子によってディスエーブルにされなければならない．

(3)　$\overline{\text{RQ}}/\overline{\text{GT}}$の動作

$\overline{\text{RQ}}/\overline{\text{GT}}$シーケンスの詳細なタイミングを**図8-39**に示す．

$\overline{\text{RQ}}/\overline{\text{GT}}$ラインによってバス・コントロールの移動を要求するためには，素子は1CPUクロック期間以上ラインをローに駆動する必要はない．これはリクエスト・パルスを構成する．$\overline{\text{RQ}}/\overline{\text{GT}}$ラインをサンプルするCPUのクロック・エッジに関して，適当なセットアップとホールドのタイムを保証するために，リクエスト・パルスはCPUクロックと同期していなければならない．

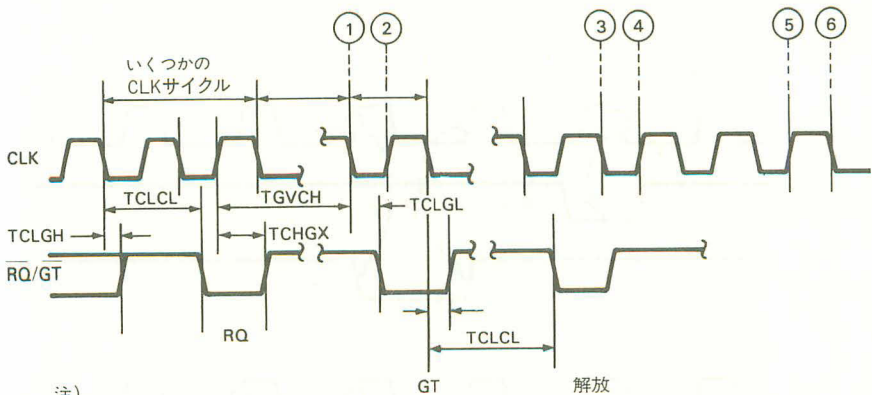
リクエスト・パルスの発行後，次のローからハイへのクロック・エッジから始まる，許

注）
① 8086は，このエッジでパッシブ・ステートから$\overline{\text{S2}}$，$\overline{\text{S1}}$，$\overline{\text{S0}}$をフロートにする．
② 8086は，このエッジでアドレス/ステータス/データ・バス，$\overline{\text{RD}}$と$\overline{\text{LOCK}}$をフロートにする．
③ 他のマスターは，このエッジでパッシング・ステートから$\overline{\text{S2}}$，$\overline{\text{S1}}$，$\overline{\text{S0}}$をフロートにする．
④ 他のマスターは，このエッジでアドレス/ステータス/データ・バス，$\overline{\text{BHE}}$，そして$\overline{\text{LOCK}}$をフロートにする．
⑤ 8086は，コントロール・ラインを再び駆動する．
⑥ 8086は，アドレス/ステータス/データのラインを再び駆動する．

図8-39　要求/許可のシーケンス

可のパルスのサンプリングを素子が開始しなければならない．8086 はリクエストにすぐ続くクロック期間に許可のパルスで応答できるので，$\overline{RQ}/\overline{GT}$ラインはリクエストと許可のパルスの間に正のレベルに戻らない．したがって，エッジ・トリガ・ロジックは許可パルスを捕えることができない．また，リクエスト・パルスを発生する回路はＣＰＵからの許可を検出する時間内にリクエストを取り除くことの保証が必要である．許可のパルスを受けた後，要求元素子はローカル・バスを駆動する．要求元マスタが許可を待つのを開始するのに用いる，アドレスまたはデータのバス，$\overline{LOCK}$または$\overline{RD}$をクロックのエッジまで 8086 はフロートにしない．したがって，要求元マスタはローカル・バスを駆動する前に8086 のフロート・ディレイ・タイム（TCLAZ アドレス・フロートまたは TCHDZ データ・フロート）の間，待たなければならない．この予防策により，要求元マスタがローカル・バスのアクセスを得る間のバス競合を防止する．

8086 にローカル・バスのコントロールを返すために，別のバス・マスタは同じ$\overline{RQ}/\overline{GT}$ラインにレリーズ・パルスを出す．8086 は，レリーズ・パルスを検出して 3 クロック・サイクル後に$\overline{S0}$-$\overline{S2}$のステータス・ラインを駆動する．8086 は，ステータス・ラインが駆動されて TCHCL（クロックがハイの時間）の後にアドレス／データ・バスを駆動する．別のバス・マスタは，8086 がバスのコントロールを再び得るときまでに，ローカル・バスをフロート状態にして，8288 やアドレス・ラッチなどの他のインターフェイス回路を再びイネーブルとしなければならない．要求元素子は許可のパルスを受けて少なくとも 1 クロック・サイクル後までレリーズ・パルスを発行せず，以前のレリーズ・パルスの少なくとも 1 クロック・サイクル後まで新しいリクエストを発行してはならない．

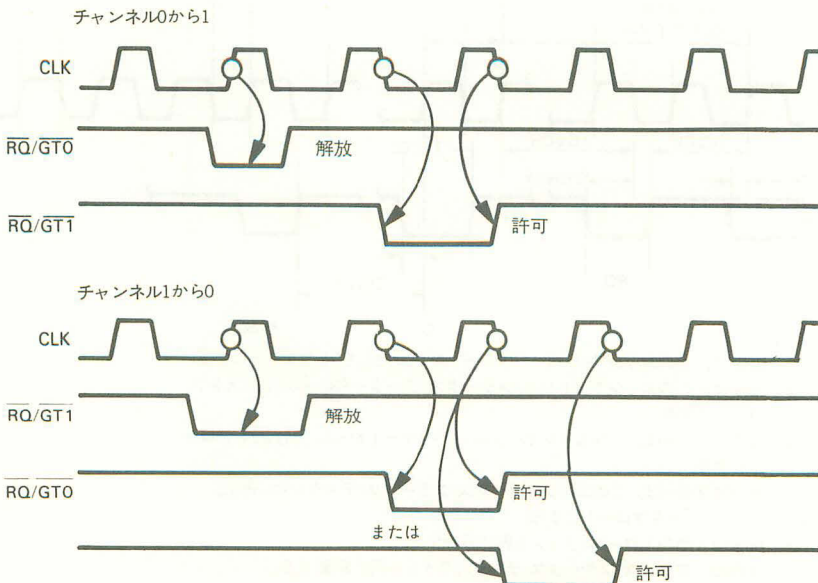



図8-40　チャンネル間のディレイ

(4)　$\overline{\mathrm{RQ}}$／$\overline{\mathrm{GT}}$の潜伏

単一の$\overline{\mathrm{RQ}}$／$\overline{\mathrm{GT}}$ラインでの$\overline{\mathrm{RQ}}$から$\overline{\mathrm{GT}}$へのディレイは，ＨＯＬＤからＨＬＤＡへのディレイに類似している．ミニマム・モードの8086に対して与えられたケースはまた，マキシマム・モードにも適用される．各ケースにおいて，8086による$\overline{\mathrm{RQ}}$の検出から要求元マスタによる$\overline{\mathrm{GT}}$の検出に対するディレイは，(ＨＯＬＤからＨＬＤＡへのディレイ)－(ＴＨＶＣＨ＋ＴＣＨＣＬ＋ＴＣＬＨＡＶ）である．これは，アイドルのバス・インターフェイスでのクロック期間の最大ディレイである．他のすべての場合，ディレイはミニマム・モードの結果から476nsを引いたものに等しい．8086が前に$\overline{\mathrm{RQ}}$／$\overline{\mathrm{GT}}$ラインの１つで許可を発行しているならば，最初の素子がその$\overline{\mathrm{RQ}}$／$\overline{\mathrm{GT}}$ラインにレリーズ・パルスでインターフェイスを解放するまで，他の$\overline{\mathrm{RQ}}$／$\overline{\mathrm{GT}}$ラインのリクエストは許可を受けられない．１つの$\overline{\mathrm{RQ}}$／$\overline{\mathrm{GT}}$ラインの解放から他のラインの許可までのディレイは一般に，**図8-40**に示しているように１つのクロック期間である．

しばしば，$\overline{\mathrm{RQ}}$／$\overline{\mathrm{GT1}}$の解放から$\overline{\mathrm{RQ}}$／$\overline{\mathrm{GT0}}$の許可までのディレイは，２つのクロック期間を要し，8086のエグゼキューション・ユニットからのコントロール移動のための保留状態のリクエストの関数となる．インターフェイスが他の$\overline{\mathrm{RQ}}$／$\overline{\mathrm{GT}}$ラインでバス・マスタのコントロール下にあるとき，要求から許可までのディレイは他のバス・マスタの関数である．プロトコルには，8086が別のバス・マスタを強制的にバスから切り離せるよ

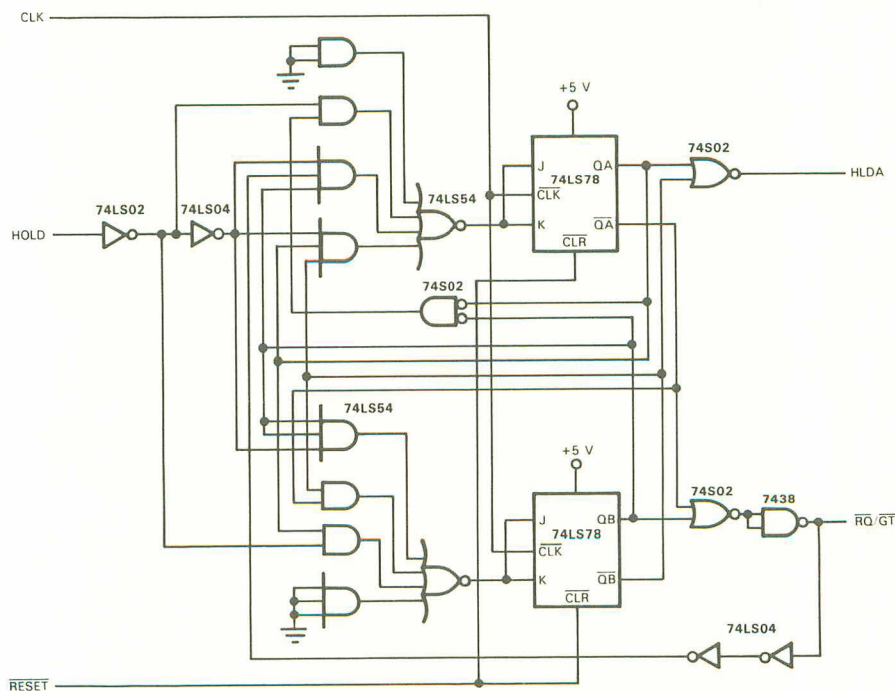



図8-41　ＨＯＬＤ／ＨＬＤＡから$\overline{\mathrm{RQ}}$／$\overline{\mathrm{GT}}$への変換回路

うな機構は含まれていない．誤って別のバス・マスタがシステムをハング状態にしないことを確実にするためには，ウオッチドッグ・タイマを用いなければならない．

(5) HOLD／HLDAから$\overline{RQ}$／$\overline{GT}$への変換

HOLD／HLDAハンドシェイク·シーケンスを$\overline{RQ}$／$\overline{GT}$パルス·シーケンスに変換する回路を**図8-41**に示す．

許可パルスを受けた後，8086がローカル・バスを切り離すTCHCL(min) 前に，HLDAはイネーブルになる．要求元回路がHLDAの20ns以内にバスを駆動するならば，1クロックの期間だけアクノリッジ・パルスを遅らせることが望ましい．HLDAは，HOLDがディスエーブルとなって1クロック期間以内に落とされる．HLDAはまた，要求元マスタがステータス・ラインのコントロールを放棄するための2TCLCL＋TCLCHと残りの信号をフロートとするための3TCLCLを与えるために，レリーズ・パルスの開始で落ちる．

第9章

Multibus

Multibusは汎用マルチプロセッシング・システム・バスである．この標準的バスは，機械的，電気的，そして形式の仕様を有する．Multibusは，インテルiSBCシングル・ボード・マイクロコンピュータの製品に用いられている．Multibusとコンパチブルな製品はまた，他の製造業者によっても提供される．マルチプロセッシング・システムを設計する人は，次の2つの重要な理由からMultibusを用いて自分のシステムを構成すべきである．

1．新しいシステム開発に関連した時間とコストの節約のために．

2．Multibusに利用できる広範な種類の製品との互換性を得るために．

8086がマキシマム・モードで構成されるとき，8288バス・コントローラと8289バス・アービタは，Multibusシステム・バスの電気的特性とAC特性と完全に互換性のあるバス・アクセスとコントロールのインターフェイスを備えている．ミニマム・モードで構成されるとき，マルチプロセッサ・システムが必要でなければ（適当な信号をエンコードするための何か外部のロジックを有するが），8086はMultibusで容易に動作できる．すべてのマルチプロセッサ・システムでは，8086はマキシマム・モードで構成される必要がある．

Multibusは，広範な種類の計算モジュールと同等に用いることの可能な，融通性のあるコミュニケーション・チャンネルを備えている．システムにおけるモジュールは，マスタあるいはスレーブである．マスタはバスの利用を得てデータ転送を開始するが，スレーブは単にデータ転送を行なう．バスは，システムで8ビットと16ビットのマスタとスレーブの混合を許している．バスは，16データ・ライン，20アドレス・ライン，8インタラプト・ライン，さらにコントロールとバス判定のラインをサポートしている．他のラインには，パワー・バス，パワー・バックアップ，そしてメモリをバッテリー・バックアップ・システムに切り替えるためのパワー・センス信号が含まれる．Multibusでのピン割当ての完全な一覧を**表9-1**と**9-2**に示す．以下に信号の機能的な記述を示す．

表9-1　マルチバス・ボードP1コネクタのバス信号のピン割当て

	ピン	(素子側)		ピン	(回路側)	
		ニーモニック*	説明		ニーモニック	説明
パワー・サプライ	1	GND	信号GND	2	GND	信号GND
	3	+5V	+5Vdc	4	+5V	+5Vdc
	5	+5	+5Vdc	6	+5	+5Vdc
	7	+12V	+12Vdc	8	+12V	+12Vdc
	9	−5V	−5Vdc	10	−5V	−5Vdc
	11	GND	信号GND	12	GND	信号GND
バス・コントロール	13	BCLK/	バス・クロック	14	INIT/	イニシャライズ
	15	BPRN/	バス・プライオリティ入力	16	BPRO/	バス・プライオリティ出力
	17	BUSY/	バス・ビジィ	18	BREQ/	バス・リクエスト
	19	MRDC/	メモリ・リード・コマンド	20	MWTC/	メモリ・ライト・コマンド
	21	IORC/	I/Oリード・コマンド	22	IOWC/	I/Oライト・コマンド
	23	XACK/	トランスファー・アクノリッジ	24	INH1/	RAMディスエーブルのインヒビット1
バス・コントロールとアドレス	25		予備	26	INH2/	PROMまたはROMディスエーブルのインヒビット2
	27	BHEN/	バイト・ハイ・イネーブル	28	AD10/	
	29	CBRQ/	コモン・バス・リクエスト	30	AD11/	
	31	CCLK/	一定クロック	32	AD12/	アドレス・バス
	33	INTA/	インタラプト・アクノリッジ	34	AD13/	
インタラプト	35	INT6/	パラレル・インタラプト・リクエスト	36	INT7/	パラレル・インタラプト・リクエスト
	37	INT4/		38	INT5/	
	39	INT2/		40	INT3/	
	41	INT0/		42	INT1/	
アドレス	43	ADRE/	アドレス・バス	44	ADRF/	アドレス・バス
	45	ADRC/		46	ADRD/	
	47	ADRA/		48	ADRB/	
	49	ADR8/		50	ADR9/	
	51	ADR6/		52	ADR7/	
	53	ADR4/		54	ADR5/	
	55	ADR2/		56	ADR3/	
	57	ADR0/		58	ADR1/	
データ	59	DATE/	データ・バス	60	DATF/	データ・バス
	61	DATC/		62	DATD/	
	63	DATA/		64	DATB/	
	65	DAT8/		66	DAT9/	
	67	DAT6/		68	DAT7/	
	69	DAT4/		70	DAT5/	
	71	DAT2/		72	DAT3/	
	73	DAT0/		74	DAT1/	
パワー・サプライ	75	GND	信号GND	76	GND	信号GND
	77		予備	78		予備
	79	−12V	−12Vdc	80	−12V	−12Vdc
	81	+5V	+5Vdc	82	+5V	+5Vdc
	83	+5V	+5Vdc	84	+5V	+5Vdc
	85	GND	信号GND	86	GND	信号GND
すべてのニーモニックはIntel Corporationが版権を所有（1978）						

* 負が真（アクティブ・ロー）の信号に対してこの本では，信号名の上に線を引くかあるいは信号名の後に斜線を引く（例　$\overline{\text{BUSY}}$=BUSY/）２つの記号を用いている．

表9-2 オプショナル・バス信号のP2コネクタピン割当て

ピン	(素子側) ニーモニック	説明	ピン	(回路側) ニーモニック	説明
1	GND	信号	2	GND	信号GND
3	V_{CCB}	+5Vバッテリー	4	V_{CCB}	+5V バッテリー
5		予備	6	V_{CCPP}	+5V パルス化パワー
7	V_{BBB}	−5Vバッテニー	8	V_{BBB}	−5V バッテリー
9		予備	10	予備	+
11	V_{DDB}	+12Vバッテリー	12	V_{DDB}	+12V バッテリー
13	PFSR/	パワー・フェイル・センス・リセット			
			14	予備	+
15	V_{AAB}	−12Vバッテリー	16	V_{AAB}	−12V バッテリー
17	PFSN/	パワー・フェイル・センス	18	ACLO	AC ロー
19	PFIN/V	パワー・フェイル・インタラプト	20	MPRO/	メモリ・プロテクト
21	GND	信号GND	22	GND	信号 GND
23	+15V/	+15V	24	+15V	+15V
25	−15V/V	−15V	26	−15V	−15V
27	PAR1/	パリティ1	28	HALT	バス・マスタ HALT
29	PAR2/	パリティ2	30	WAIT/	バス・マスタ WAIT STATE
31			32	ALE	バス・マスタALE
33			34	予備	
35			36	予備	
37			38	AUX RESET/	リセット・スイッチ
39			40		
41			42		
43			44		
45	予備		46		
47			48	予備	
49			50		
51			52		
55			56		
57			58		
59			60		

注)
1. PFINはスレーブ・モジュール上でもし可能ならば,P1のINT0/への接続を選択できるべきである.
2. 未定義のピンはすべて将来のための予備である.
すべてのニーモニックは Intel Corporation が版権を所有 (1978).

9.1 初期化信号ライン

(1) $\overline{\text{INIT}}$

初期化信号は,前もって決められた状態にシステム全体をリセットする. $\overline{\text{INIT}}$ は,バス・マスタの1つあるいは外部ロジックによって供給される.

9.2 アドレスとインヒビットのライン

(1) $\overline{\text{ADR0}}$ − $\overline{\text{ADR13}}$

20のアドレス・ライン が, アクセスされるメモリ位置あるいはI/Oポートのアドレスを

伝送するために用いられる．$\overline{\mathrm{ADR13}}$は最上位ビットであり，$\overline{\mathrm{ADR0}}$は最下位ビットである．8ビットのバス・マスタは，メモリのアドレス指定に16のアドレス・ライン（$\overline{\mathrm{ADR0}}$－$\overline{\mathrm{ADRF}}$）を，I/Oポートの選択に8のアドレス・ライン（$\overline{\mathrm{ADR0}}$－$\overline{\mathrm{ADR7}}$）を用いる．16ビットのバス・マスタは，20のアドレス・ラインでメモリのアドレス指定を，下位12のアドレス・ライン（$\overline{\mathrm{ADR0}}$－$\overline{\mathrm{ADRB}}$）でI/Oポートの選択を行なう．しかし8088は，8ビット・バスのCPUと考えられるが，20のアドレス・ラインすべてを用いることができる．

(2) $\overline{\mathrm{INH1}}$

インヒビットRAM信号は，RAMメモリ素子がアドレス・バス上のアドレスに応答するのを防止する．$\overline{\mathrm{INH1}}$によって，ROMとRAMのメモリが同じメモリ領域に割り当てられているときに，ROMメモリ素子がRAM素子を無効にすることができる．

(3) $\overline{\mathrm{INH2}}$

インヒビットROM信号は，ROMメモリ素子がアドレス・バス上のアドレスに応答するのを防止する．$\overline{\mathrm{INH2}}$によって，ROMと補助ROMのメモリが同じメモリ領域に割り当てられているときに，補助ROMがROM素子を無効にすることができる．

$\overline{\mathrm{INH1}}$と$\overline{\mathrm{INH2}}$はまた，メモリ・マップドI/O素子がRAMとROMの素子をそれぞれ無効にできるようにするためにも用いられる．

(4) $\overline{\mathrm{BHEN}}$

$\overline{\mathrm{BHEN}}$は，データがMultibusの上位8のデータ・ラインで伝送されることを示すのに用いられる．この信号は，16ビットのメモリまたはI/Oモジュールを用いるシステムで使用される．

9.3 データ・ライン

(1) $\overline{\mathrm{DAT0}}$－$\overline{\mathrm{DATF}}$

16の両方向データ・ラインが，メモリ位置あるいはI/Oポートとの情報交換に用いられる．$\overline{\mathrm{DATF}}$は最上位ビットであるが，8ビットのシステムでは$\overline{\mathrm{DAT0}}$－$\overline{\mathrm{DAT7}}$のラインだけが用いられ，$\overline{\mathrm{DAT7}}$が最上位ビットになる．$\overline{\mathrm{DAT0}}$は常に最下位ビットである．

9.4 バス競合解決ライン

(1) $\overline{\mathrm{BCLK}}$

バス・クロックの負のエッジは，バス競合の同期化に用いられる．$\overline{\mathrm{BCLK}}$はCPUクロックとは非同期である．$\overline{\mathrm{BCLK}}$は，遅くなったり，停止したり，あるいはデバッグの間でシングル・ステップになる．$\overline{\mathrm{BCLK}}$はCPUクロックとは非同期である．

(2) $\overline{\mathrm{CCLK}}$

一定クロックは，特に示されていない一定周波数のクロック信号を供給する．

(3) $\overline{\mathrm{BPRN}}$

バス・プライオリティ入力信号は，より高い優先権の素子がシステム・バスの使用を要

求していないことをバス・マスタに通知する．$\overline{\mathrm{BPRN}}$は$\overline{\mathrm{BCLK}}$と同期している．この信号は，シリアルの優先権判定を用いるならば，“ディジィ・チェーン”となる．パラレルの優先権判定を用いるときは，バス・アービタが$\overline{\mathrm{BPRN}}$を生成する．

(4) $\overline{\mathrm{BPRO}}$

これはバス・プライオリティ出力信号である．$\overline{\mathrm{BPRN}}$と同様に，シリアルの優先権判定が用いられるときは“ディジィ・チェーン”となり，$\overline{\mathrm{BPRO}}$は次に低い優先権のモジュールの$\overline{\mathrm{BPRN}}$入力に加えられる．パラレルの優先権判定を用いるときは，バス・アービタがこの信号を供給しなければならない．$\overline{\mathrm{BPRO}}$は$\overline{\mathrm{BCLK}}$と同期している．

(5) $\overline{\mathrm{BUSY}}$

バス・ビジィ信号は，システム・バスが使用されていることを示すために，現在のバス・マスタによって供給される．$\overline{\mathrm{BUSY}}$は，システム・バスのコントロールを得られるかどうかを判断するために，他の素子によって用いられる．$\overline{\mathrm{BUSY}}$は$\overline{\mathrm{BCLK}}$と同期している．

(6) $\overline{\mathrm{BREQ}}$

バス・リクエスト信号は，バス・マスタになることの要求を示すために，素子によって用いられる．$\overline{\mathrm{BREQ}}$は$\overline{\mathrm{BCLK}}$と同期している．

(7) $\overline{\mathrm{CBRQ}}$

$\overline{\mathrm{CBRQ}}$は，現在のバス・マスタに他のマスタのバス使用の要求を知らせるために，可能性のあるすべてのバス・マスタによって用いられる．$\overline{\mathrm{CBRQ}}$がハイならば，現在のバス・マスタは，他の素子がバスを要求していなくて，現バス・マスタがバスを保持すべきであることを知る．

9.5 情報伝送プロトコルのライン

システム・バスのコントロールを有するバス・マスタは，データ伝送コントロール信号のすべてを発生する．すべてのアドレス信号（書き込みが行なわれるときはデータ信号も）伝送コントロール信号パルスの少なくとも50ns前で安定でなくてはならず，コントロール信号パルスが取り除かれてから少なくとも50ns後までの間は有効でなければならない．

情報伝送プロトコルのラインは，$\overline{\mathrm{BCLK}}$と同期していない．

(1) $\overline{\mathrm{MRDC}}$

メモリ・リード・コントロールは，メモリ位置のアドレスがアドレス・ライン上に設定されていて，そのアドレス位置の内容がデータ・ライン上に設定されるべきであることを示す．

(2) $\overline{\mathrm{MWTC}}$

メモリ・ライト・コントロールは，メモリ位置のアドレスがアドレス・ライン上に設定されていて，データがシステム・データ・ライン上に設定されていることを示し，このデータはアドレス指定されたメモリ位置に書き込まれなければならない．

(3) $\overline{\mathrm{IORC}}$

I/O リード・コントロールは，入力ポートのアドレスがシステム・アドレス・ライン上に設定されていて，入力ポートのデータがデータ・ライン設定されるべきであることを示す．

(4) $\overline{\mathrm{IOWC}}$

I/Oライト・コントロールは，出力ポートのアドレスがシステム・アドレス・ライン上に設定されていて，データがシステム・データ・ライン上に設定されていることを示し，このデータはアドレス指定されたポートに出力されなければならない．

(5) $\overline{\mathrm{XACK}}$

すべての交換にはハンドシェイクを伴う．したがって，選択されたバス・スレーブは移動コントロール信号に応じて，バス・マスタにアクノリッジ信号を供給しなければならない．トランスファ・アクノリッジ信号は，指定された動作の完了を示すための必要な応答である．

(6) $\overline{\mathrm{AACK}}$

アドバンスド・アクノリッジ信号は，8080Aマイクロプロセッサで用いられる．$\overline{\mathrm{AACK}}$は，CPUがウエート状態に入ることなく指定された動作の終了を可能とする先行のアクノリッジである．$\overline{\mathrm{AACK}}$を供給するバス・スレーブはまた，$\overline{\mathrm{XACK}}$も供給しなければならない．この必要条件は，すべてのバス・マスタが$\overline{\mathrm{AACK}}$信号に応答するとは限らないので，満たされる必要がある．

9.6 非同期インタラプト・ライン

(1) $\overline{\mathrm{INT0}}$ - $\overline{\mathrm{INT7}}$

この8つのプライオリティ・インタラプト・リクエスト・ラインは，パラレルのインタラプト解決回路で用いられる．$\overline{\mathrm{INT7}}$ が最も低く，$\overline{\mathrm{INT0}}$ が最も高い優先権を持つ．

(2) $\overline{\mathrm{INTA}}$

$\overline{\mathrm{INTA}}$ は，外部ロジックがMultibusデータ・ラインにインタラプト・ベクタの情報を設定することを要求するために，バス・マスタによって用いられる．

9.7 パワー供給ライン

各種の安定化されたパワーを供給するラインがバスに備わっている．各モジュールは，大容量コンデンサと，存在するロジック素子に局所的な高周波用コンデンサとを共に備える必要がある．

9.8 予備ライン

予備ラインは用いてはならず，将来におけるIntelの定義に利用可能な状態にして置く必要がある．

Multibusは論理的に，8086の分離バスと類似している．アドレスは，アドレス・ライン$\overline{\mathrm{ADR0}}$から$\overline{\mathrm{ADR13}}$で供給される（なお，アドレス・ラインのナンバーは16進数である）．

Multibusは，リード・コントロール信号の選択された素子への伝達が可能となる前に有効アドレスからの50nsのディレイを必要とする．リード・コントロール・パルスは少なくとも100nsの幅が必要であり，アドレスはリード・コントロール信号が終了して少なくとも50ns後も安定でなければならない．選択された素子が，100nsのリード信号あるいは指定されたアドレス・アクセス・タイムの最小値150ns以上を要するならば，素子は$\overline{\mathrm{XACK}}$（トランスファ・アクノリッジ）信号を用いてリード・サイクルを拡張できる．この信号は，8284 RDY入力に接続されるレディ信号に同等である．$\overline{\mathrm{XACK}}$は通常“ノット・レディ”で，素子がデータの受信あるいは送信に対してレディであり，バス・サイクルの終了が可能であることをCPUに通知するために，アクティブに駆動される．異なるタイプのCPUが混合したマルチプルプロセッサ・システムにおいて選択された素子の独立な動作を可能とするために，Multibusでは，読み込みまたは書き込みのコントロール信号に対してではなく$\overline{\mathrm{XACK}}$信号に関してデータのセットアップとホールドのタイムを指定している．

ライト・バス・サイクルはリード・バス・サイクルに類似している．書き込まれるデータは，ライト・コントロール信号より最小50ns前に有効でなければならず，ライト・コントロール信号に続いて最小50nsは有効に維持されなければならない．

Multibusに付属のマスタ・モジュールは，最小のセットアップとホールドのタイムあるいはコントロール・パルス幅に違反してはならない．多くの設計では，最大帯域幅で動作させたときに最小の余裕よりも良い値が得られる．スレーブ・モジュールは，最小のセットアップとホールドのタイムを認められる必要があるが，適当な時間だけ$\overline{\mathrm{XACK}}$を返すのが遅れれば，アクセス・タイムを拡張する．

Multibusは，2つの基本的なインタラプト処理の方法を備えている．それは次のものである．

1．インタラプト・ベクタがバスで転送されない方法．インタラプト・ベクタはバス・マスタのインタラプト・コントローラによって生成される．割り込みを要求するスレーブは，バス・マスタとして同じモジュールの一部でなければならない．要求元スレーブが他のモジュールの一部ならば，スレーブは割り込みの要求のためにMultibusのインタラプト・リクエスト・ライン（$\overline{\mathrm{INT\,0}}$－$\overline{\mathrm{INT\,7}}$）を用い，この割り込みはバス・マスタのインタラプト・コントローラによって処理される．

2．インタラプト・ベクタがバスで転送される方法．スレーブ素子が割り込みを要求すると，インタラプト・コントロール・ロジックはプロセッサに割り込みをかける．プロセッサは，$\overline{\mathrm{INTA}}$ラインをローにしてシステム・バスをロックして割り込みを受け付ける．これによりインタラプト・ベクタの転送を可能にする．最初の$\overline{\mathrm{INTA}}$サイクルに続いて，インタラプト・コントロール・ロジックは，現在割り込みを要求している最も高い優先権のスレーブのアドレスを決定する．このアドレスは，アドレス・バス上に設定される．アドレス指定されたスレーブは，インタラプト・ベクタ・アドレスをマスタに伝送して応答する．

マスタとスレーブのモジュール設計に，標準の非同期データ伝送プロトコルとタイミングの仕様を与えてさらに，Multibusは複数のマスタがバス・コントロールの交換に用いる

標準的プロトコルを与える．非同期のマスタのバス共有を可能にするために，Multibusは，それに接続されるモジュールに局所的なクロック信号に独立なそれ自身のクロックを維持している．$\overline{\text{BCLK}}$で表わされるMultibusクロックは，バス・アクセスに対する非同期の要求の同期をとる．これにより，判定ロジックの優先権解決と一度に1つのマスタのアクセスを許可することが可能となる．

このバス判定法で，種々の速度で動作するマスタがシステム・バスの利用に対して公平に競うことができる．しかしマスタがシステム・バスのコントロールを獲得すると，転送速度は，Multibusクロック信号$\overline{\text{BCLK}}$ではなく，マスタの能力とそれに関連したスレーブ・モジュールだけに依存する．最小のアドレスのセットアップとホールドのタイムそして最小のコントロール・パルス幅を考慮すると，最大のバス帯域幅は5MHzである．バス判定と一般的なメモリ応答時間の付加的なオーバヘッドを考えると，実際の転送速度はしばしば2MHzのオーダになる．

Multibusは，バス・マスタ間でのシリアルまたはパラレルの優先権判定を可能にする．シリアル・プライオリティの方法を**図9-1**に示す．

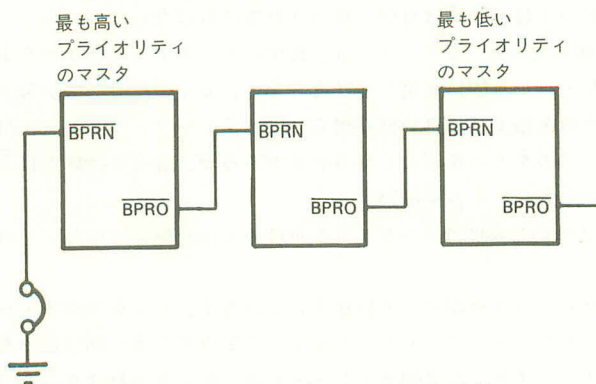



図9-1　シリアル・プライオリティの方法

最も高いプライオリティのマスタは$\overline{\text{BPRN}}$をグランドに接続する．各マスタのプライオリティ・イネーブル出力は，次に低いプライオリティのマスタのプライオリティ入力$\overline{\text{BPRN}}$に接続される．そのマスタがバスを必要としなければ，$\overline{\text{BPRN}}$を$\overline{\text{BPRO}}$に伝える．これは上位のプライオリティを下位のマスタに伝える．バスを必要とするマスタは$\overline{\text{BPRO}}$にハイを出力する．これは下位レベルのマスタに対するプライオリティを否定する．このロジックは，システムにおける最も高いプライオリティのマスタからのプライオリティ・イネーブルを最も低いものへとディジィ・チェーンとして伝える．より高いプライオリティのマスタは現在のバス・マスタがすでに進行中のバス・サイクルを終了するまで待たなければならないので，他の信号はMultibusのアイドルまたは“ノット・ビジィ”の状態を示すために含まれる必要がある．この機能を果たす$\overline{\text{BUSY}}$ラインは，すべてのバス・マスタに共通の信号である．Multibusを用いているマスタは，バスへのアクセスを得るためのその能力を決めるために，$\overline{\text{BUSY}}$の時間を記録している．

$\overline{\text{BCLK}}$のハイからローへの遷移におけるサンプルで，$\overline{\text{BPRN}}$がロー（アクティブ）で，$\overline{\text{BUSY}}$がハイ（インアクティブ）であり，アイドル・バスを表わしているならば，現在のマスタがその伝送を終了する前により高いプライオリティのマスタがバスのコントロールを獲得するのを防ぐために，現在のマスタは$\overline{\text{BCLK}}$の次のハイからローへの遷移の前に$\overline{\text{BUSY}}$をローに駆動しなければならない．

現在のマスタがプライオリティを失なうと，伝送終了後に$\overline{\text{BUSY}}$を解放してMultibusに対する接続をフロートにしなければならない．

他のマスタがバスの要求を必要とするならば，$\overline{\text{BCLK}}$のハイからローへの遷移の前に低いプライオリティのマスタをディスエーブルにしなければならない．さもなければ，プライオリティ解決回路とより低いプライオリティのマスタにおいて競走条件が生じる．また，より高いプライオリティのマスタからのプライオリティ・ディスエーブルはディジィ・チェーンを通してより低いプライオリティのマスタに伝えられる必要があるので，最高から最低のプライオリティのマスタへの全体の伝播ディレイは1つの$\overline{\text{BCLK}}$クロック期間を越えてはならない．これは，シリアルのプライオリティ判定が適応可能なマスタの数に上限を設定する．

パラレルのプライオリティ判定は，$\overline{\text{BCLK}}$のハイからローへの遷移で各マスタがバス・

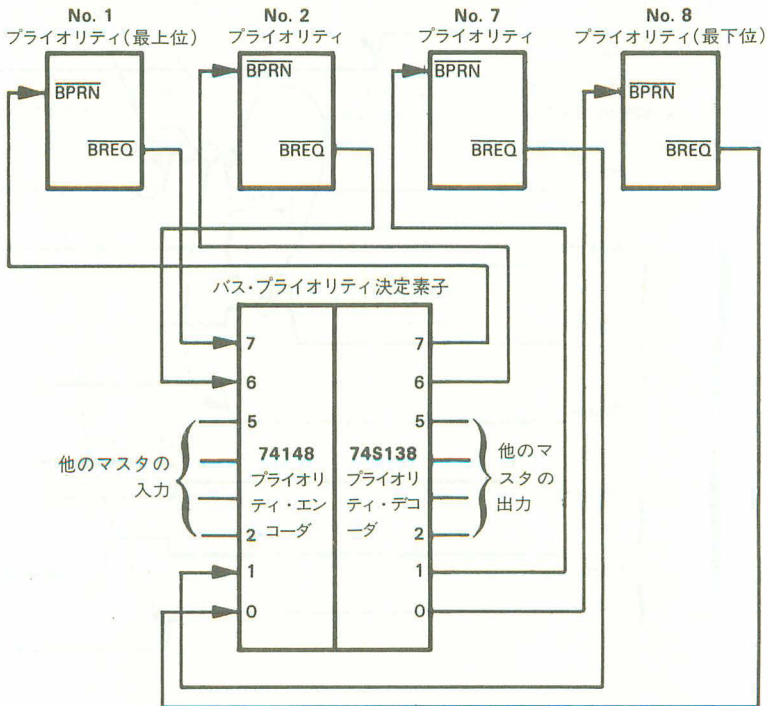



図9-2 パラレルのプライオリティ決定とバス交換タイミング

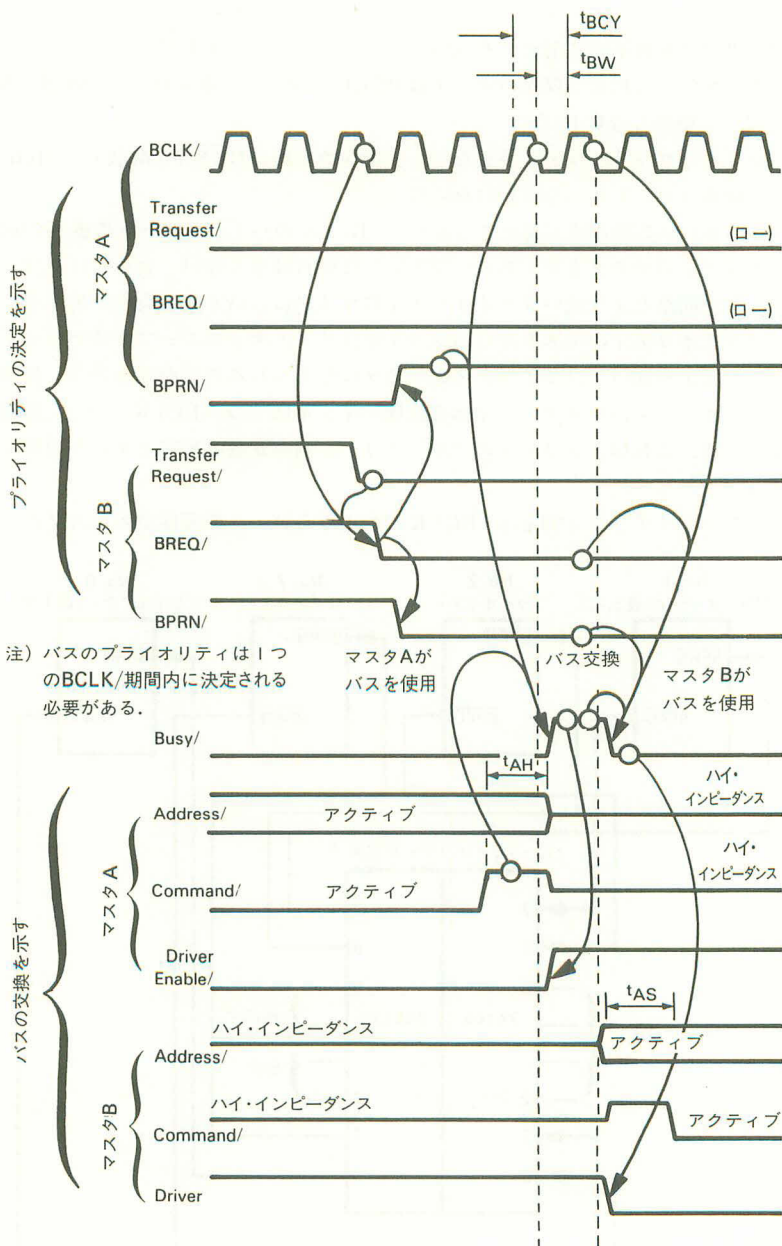



図9-2　パラレルのプライオリティ決定とバス交換のタイミング（続き）

リクエスト$\overline{\text{BREQ}}$を発行し，外部のユーザ定義による回路がプライオリティを解決することを除けば，シリアルのプライオリティ判定に類似している．プライオリティの決定回路は，$\overline{\text{BCLK}}$の1期間内に各マスタに対して解決を行ない，安定なプライオリティの入力，$\overline{\text{BPRN}}$を戻さなければならない，パラレルの決定回路とバス交換のタイミングの例を図9-2に示す．

9.9 Multibus 構成の概念

Multibus構成は，マルチプロセッサ・システムに対して定義の明確なバス構造を与える．バスは，マルチプロセッサ間の資源の共有とシステムのプロセッサ間の通信の手段として役立つ．マルチプロセッサ・システムには次の2つの基本的な形式が存在する．

1．密結合システム．密結合のシステムでは，複数のプロセッサは共通のメモリ領域を通して互いに情報を交換することによって通信を行なう．
2．疎結合システム．疎結合のシステムでは，複数のプロセッサはI/O構造を通して互いに情報を交換することによって通信を行なう．一般に，シリアルのコミュニケーション・リンクが用いられる．しかし，ある場合には情報は高速ディスクなどのマス記憶素子を通して伝えられる．

Multibusは密結合システムの必要条件を満たすように設計されているにもかかわらず，マルチＣＰＵ処理システムがBisyncまたはHDLCに似た標準I/Oコミューニケーション・プロトコルによって疎結合となることが可能である．バス共有の機構は，Multibusプロトコルとユーザ定義のプライオリティ決定回路の組合せから成る．Multibusは，バスのリクエストのために，プロセッサがバスのプライオリティを持つときに確認を受け取るために，そしてバスの有効性を示すために，各プロセッサに対して基本的なコントロールを与える．

ユーザはタスクに最も適切なプライオリティ決定の方法を選択する必要がある．プライオリティ・システムの選択には慎重でなければならない．さもなければ，Multibusはシステムの障害となり，システム全体の性能を悪化させることになる．バスの公平な利用を確実にするために，Multibusは共通のバス・リクエスト$\overline{\text{CBRQ}}$をサポートしている．この信号により，低いプライオリティの素子がより高いプライオリティのバス・マスタからバスを要求すること，あるいはより低いプライオリティのリクエストの保留を示すことが可能になる．これは，より高いプライオリティのマスタが強制的に（プライオリティを失なわせて）現在のマスタをバスから切り離すか，より低いプライオリティのマスタが（共通バス・リクエストによって）バスを要求するまで，現在のバス・マスタがバスのコントロールを維持することを可能にする．

共通バス・リクエストに対する現在のマスタの応答は，ユーザが定義できる．これは，マスタに現在の伝送の終わりかあるいはバス動作がなければ直ちにバスを解放させるか，またはマスタは単にリクエストを無視してより高いプライオリティのマスタに対してバスのコントロールを放棄するだけである．共通バス・リクエストは設計者に，システム・バスの利用を定義するときに別のレベルのコントロールを与える．結果的に，現在のマスタ

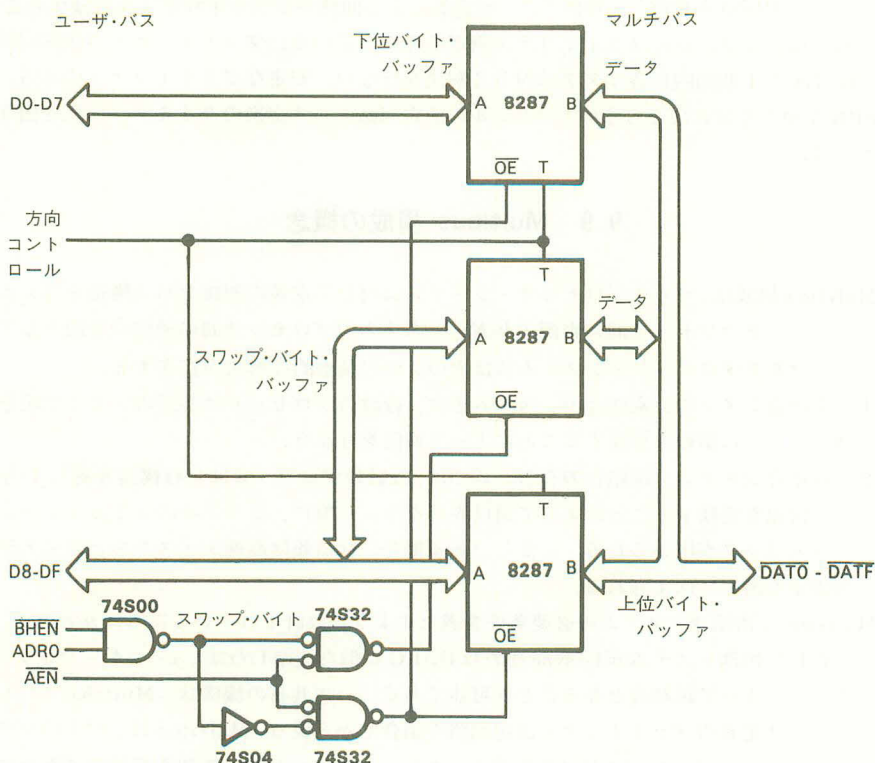



図9-3　8/16ビットの素子の伝送動作

は伝送のたびにバスの解放を強制されて再び要求することはない．これは，バスのアクセスと伝送に関したオーバヘッドを最小にする．

Multibusの$\overline{\text{BUSY}}$信号により，より高いプライオリティのマスタが強制的にプライオリティを失なわせようとしても（強制的にその$\overline{\text{BPRN}}$信号をインアクティブにする），現在のマスタはバスのコントロールを維持できる．これは，現在のマスタのバス・サイクルが他のバス・マスタに対して実行されるバス・サイクルで分離できない，複数のバス・サイクルを要する動作に必要である．現在のバス・マスタがバスのコントロールを有していれば，バスが利用できることを示す $\overline{\text{BUSY}}$ を解放せずに，バスのコントロールを続ける．この機能は，複数バス・サイクルのインタラプト・アクノリッジ・シーケンスとテスト・アンド・セットの動作に必要とされる．バスのコントロールを放棄してはならないこれらの条件を解釈するのはマスタ・モジュールの機能である．この条件が起きるときは，マスタは必要な動作の終了まで$\overline{\text{BPRN}}$を無視しなければならない．

Multibusは，アドレスとデータのラインが別々の分離システム・バスである．システムは20のアドレス・ラインで1メガバイトのアドレス領域を維持し，16ビットのデータ・バ

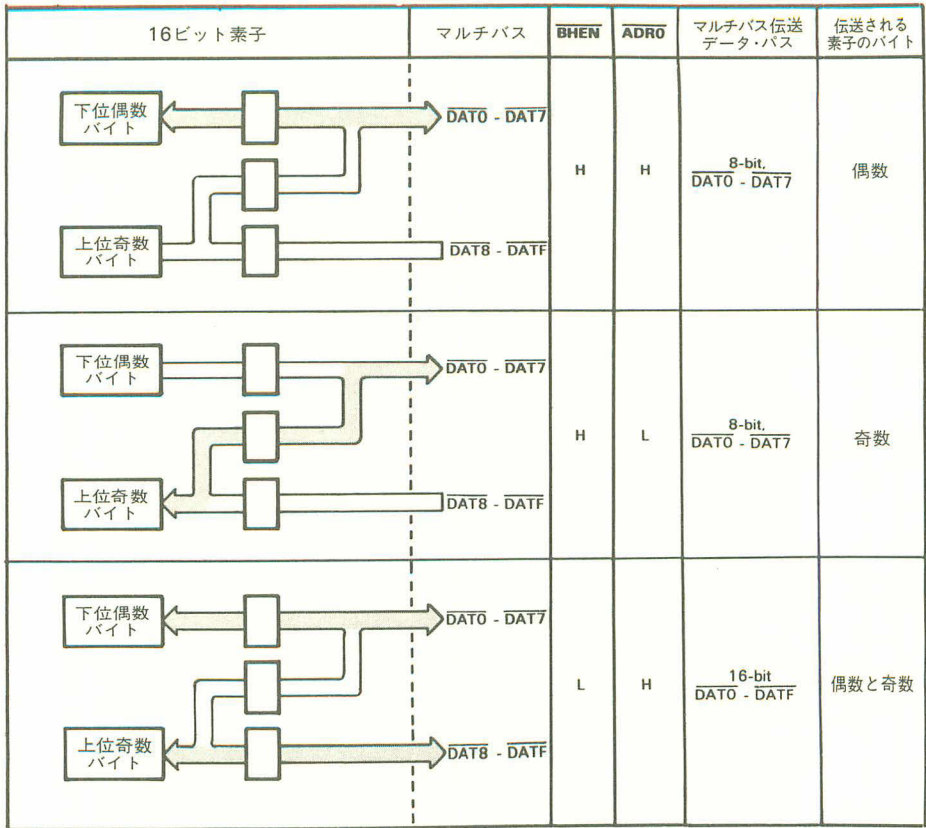



図9-3　8/16ビットの素子の伝送動作(続き)

スが存在する．8ビットのデータ・バスを維持するだけのMultibusとコンパチブルのモジュールとの互換性から，すべての8ビット伝送はアドレスを無視してデータ・バスの下位を通して行なわれる．16ビット伝送だけが16のデータ・ラインすべてを用いる．これは，データ・バスの上位と下位で上位と下位のバイトを伝送する標準的な8086の方法とは別のものである．結果的に，既存の8ビット・インターフェイスのスレーブ・モジュールに対するバイト伝送はCPUと独立している．

16ビット・インターフェイスをサポートするスレーブ・モジュールは，8ビット・データ・バスにだけインターフェイス可能なマスタ・モジュールによって一度に1バイトのアクセスができ，さらにスレーブは，16ビット・データ・バスすべてをサポートするマスタによって一度に1ワードあるいは1バイトのアクセスができる．Multibusの信号$\overline{\mathrm{BHEN}}$は，バイトあるいはワードのどちらの伝送が行なわれているかを判断するために必要な付加的情報を与える．**図9-3**に，Multibusのシステム・バスとの間でバイトとワードの情報の切り替えを$\overline{\mathrm{BHEN}}$がどのようにコントロールしているかを示す．

第10章
8086のマルチプロセッサ構成

8086ファミリーの構成要素は，シングル・プロセッサとマルチプロセッサの両構成を可能とするシステム・アーキテクチャを与えるように考案されている．この章では，マルチプロセッサ・システムにおけるマキシマム・モードの使用法について述べると共に，8086ファミリーの構成要素でサポートされる各種の構成を検討する．

8086ファミリーのマルチプロセッシングの機能は，次の２つの別々の異なる特徴に基づいている．

1．ローカル・バス上に位置し，8086ＣＰＵの基本的アーキテクチャを強調する，特殊機能のプロセッサ．
2．共通のシステム・バスを共有するマルチプルＣＰＵ．

特殊機能のプロセッサは１つのＣＰＵに寄与し，ＣＰＵの命令セットの拡張を行ない，このＣＰＵと並列に処理する．したがって，特殊機能のプロセッサはコープロセッサと呼ばれる．対照的に，マルチプル・マイクロプロセッサ・システムは，ＣＰＵの繰り返しがユーザで定義される古典的なマルチＣＰＵの環境と酷似している．システムでは２つの機能が組合される，すなわち，マルチプロセッサ・システムにおいて各ＣＰＵはそれ自身に寄与するコープロセッサを有することに注意．マルチＣＰＵのマルチプロセッサ・システムの要求は，９章で定義されているMultibusによって満たされる．

10.1　コープロセッサ

8086は，コープロセッサの利用を可能とすることを目的としたハードウエアとソフトウエアの特徴を有している．ハードウエアのサポートには，キュー・ステータスの信号（ＱＳ０，ＱＳ１），$\overline{\mathrm{TEST}}$入力，そしてＣＰＵのローカル・バスを共有するための機構（$\overline{\mathrm{RQ}}$／$\overline{\mathrm{GT}}$）が含まれる．ソフトウエアのサポートは，コープロセッサを動作させるＥＳＣＡＰＥ命令と呼ばれる特殊なクラスの命令と（ＣＰＵとコープロセッサのソフトウエアの

同期に用いる）$\overline{\text{TEST}}$入力をサンプルするＷＡＩＴ命令より成る．

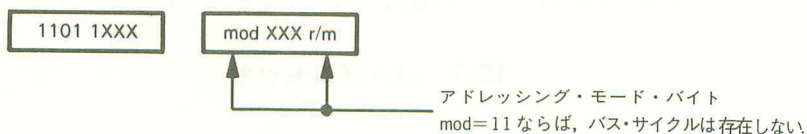
コープロセッサはＣＰＵのローカル・バスと直接にインターフェイスする．ＣＰＵのローカル・バスには，多重化されたアドレス／ステータスとアドレス／データのライン，$\overline{\text{S0}}$ $\overline{\text{S1}}$，そして$\overline{\text{S2}}$のステータス・ライン，ＱＳ０とＱＳ１のキュー・ステータス・ライン，$\overline{\text{TEST}}$，ＲＥＡＤＹ，ＲＥＳＥＴ，$\overline{\text{RQ}}$／$\overline{\text{GT}}$ラインの１つ，そしておそらく$\overline{\text{LOCK}}$のラインが含まれる．$\overline{\text{LOCK}}$信号に対する要求はコープロセッサに依存して変わる．コープロセッサは次の２つの理由から，ローカル・バス上に位置することが許されている．

１．コープロセッサはＣＰＵの動作をモニタすることができる．

２．コープロセッサはＣＰＵの資源に対して完全なアクセスを有する．

コープロセッサは，$\overline{\text{RQ}}$／$\overline{\text{GT}}$ラインの１つを通してローカル・バスのコントロールを要求する．ＣＰＵがバスのコントロールをコープロセッサに譲った後に，コープロセッサはＣＰＵとまったく同じにローカル・バス・サイクルで動作する．コープロセッサがローカル・バスの使用を終われば，バスのコントロールを得るために用いられたのと同じ$\overline{\text{RQ}}$／$\overline{\text{GT}}$ラインを通してＣＰＵにコントロールが返される．このインターフェイスの例を**図10-1**に示す．

コープロセッサは以下のシーケンスで動作状態になる．ＣＰＵが命令のフェッチを行ない命令を実行している間，コープロセッサはESCAPE 命令を待ち受けて命令の流れをモニタしている．メモリからＣＰＵの命令キューに伝送される情報は（命令でオペランドとして用いられるメモリ・データとは対照的に），命令フェッチを識別するステータス・ライン（$\overline{\text{S0}}$, $\overline{\text{S1}}$, $\overline{\text{S2}}$,）をデコードすることによって選択される．Ａ０と$\overline{\text{BHE}}$は，シングルまたはダブルのどちらのバイトの命令フェッチが実行されるかを判断するためにデコードされる．

キューより得られる情報は，キュー・ステータス・ライン（ＱＳ０，ＱＳ１）によって，命令の第１バイトあるいは後続バイトとして識別される．キュー・ステータス・ラインはまた，バイトのフェッチが行なわれないこと，あるいはキューが空であることを示す．コープロセッサのキュー（これはＣＰＵのキューに追随している）からの情報がＥＳＣＡＰＥ命令を示し，キュー・ステータスがそれを命令のオブジェクト・コードの第１バイトとして識別するならば，コープロセッサは動作を行なう．ＥＳＣＡＰＥ命令のオブジェクト・コードを次に示す．

ＸはＣＰＵに対するドント・ケアのビットであるが，コープロセッサに対して64の可能な命令を表わすことができる．ＥＳＣＡＰＥ命令に応じて，ＣＰＵは通常のアドレッシング・モード指定としてmod と r/m のフィールドを用い，指定アドレスからデータを読み込むためにバス・サイクルを実行する．コープロセッサは，選択されたメモリ位置のア

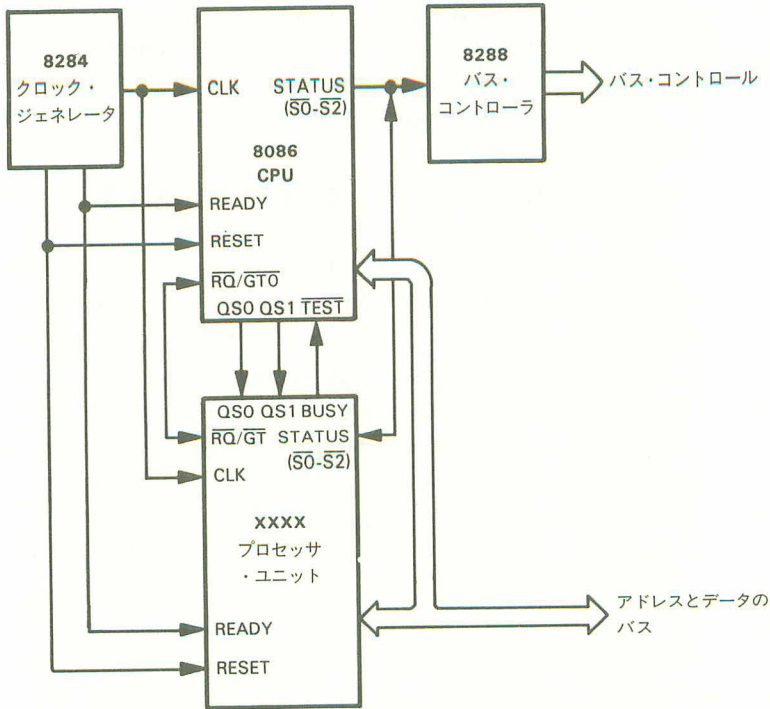



図10-1　マキシマム・モード・マルチプル・プロセッサ

ドレスとデータの両方を捕えることができる．この機構は，プログラマがメモリ・アドレッシング・モードのすべての範囲でコープロセッサに対して定義されたＥＳＣＡＰＥ命令を通常のＣＰＵ命令として取り扱うことを可能にする．１つのパラメータと共にコープロセッサにアドレスを受け渡すことができる．コープロセッサがメモリとの間で付加的データを転送する必要があれば，ＣＰＵのバスのコントロールを要求する．コープロセッサはＣＰＵのレジスタに対するアクセスを持たないので，コープロセッサで用いられるすべての情報はメモリ内に存在しなければならないことに注意．

ＣＰＵは，ＥＳＣＡＰＥ命令の間に読み込まれるデータを捕えず，リード・バス・サイクルを除いてこの命令を無効操作として取り扱う．ＥＳＣＡＰＥ命令の実行後，ＣＰＵとコープロセッサは自由に並行して特定のタスクの実行を続ける．命令実行の間，コープロセッサは一般に動作中であることを示すために$\overline{\text{ＴＥＳＴ}}$のラインをハイに保つ．プログラムでコープロセッサが動作中でないことを保証できないならば，別のＥＳＣＡＰＥ命令を実行する前に，プログラムで（可能ならば）コープロセッサのステータスを読み出すか各ＥＳＣＡＰＥ命令の前にＷＡＩＴ命令を挿入しなければならない．

ＷＡＩＴ命令は，ＥＳＣＡＰＥ命令を実行する前にコープロセッサが動作中でなくなるまで強制的にＣＰＵをウエートさせる．ＷＡＩＴ命令の間，多分，命令キューを満たすことを除いて，ＣＰＵはバスを必要としない．しかし，ＣＰＵはＷＡＩＴサイクルの間（イ

ネーブル状態ならば）割り込みに応答する．ESCAPE命令実行中にコープロセッサはCPUの命令の流れをモニタし続けていることに注意．インテルは8087コープロセッサの発表だけを行なっているが，インターフェイスの汎用性から，数値，言語のサポートなどに対する広い範囲のより高いレベルのコマンドに適用可能である．

10.2 共有システム・バスにおける多重処理

マイクロプロセッサのアプリケーションがより複雑となり，マイクロプロセッサのコストとパフォーマンスの比が小さくなるに従い，1つ以上のマイクロプロセッサでシステムを設計することがコスト的により有効になっている．マルチプロセッサ・システムは，デザインを容易に識別可能で他のシステムに対する明確な通信インターフェイスを有する機能的なサブシステムに分割することによって実現される．デザインを分割してハードウエア・インターフェイスとソフトウエア・インターフェイスを定義した後に，全体の最終結果設計時間を減少させる手段として個々のチームによって並行して，各サブシステムの設計と開発が行なわれる．この方法はすべての設計チームの間に高度な協力と調整を必要とするが，モジュラリティ，拡張性，そしてメインテナンスの容易さなどの利点がある．

システムをマルチプロセッサ分散インテリジェンス・システムに分割するとき，機能的サブシステムの必要条件が，疎結合，密結合，あるいは疎結合と密結合のプロセッサの必要性を示す．

以下の議論では，疎結合プロセッサは共有の I/O 機構を通して通信を行ない，密結合プロセッサは共有メモリを通して通信を行なう．疎結合のプロセッサに対して，各CPUは，それに付属の他の I/O 素子として取り扱われる I/O インターフェイスを通して，他のCPUと通信を行なう．このタイプのインターフェイスには，SDLC，ASYNC，GPIBなどの広範な種類の標準的プロトコルが存在する．これらのインターフェイスは，サブシステムのハードウエアが I/O インターフェイスを備えて，さらにソフトウエアがインターフェイスをコントロールして，メッセージ・プロトコルを解釈することを要する．

疎結合サブシステムは一般に，頻度の高いあるいは高速の通信を必要とせず，お互いに数マイルすら物理的に離れている．

マルチプル・プロセッサが密結合であれば，プロセッサ間でメモリを共有するための方法が与えられなければならない．8086ファミリーは，共有システム・バスへの結合によってメモリを共有する．このバスによってアクセス可能なメモリと I/O の素子は，バス・マスタとなる資格のあるすべてのCPUによって利用される．

共有のメモリと I/O の素子は，希望するプロセッサ間通信機構を与える．

マルチマスタ・システム・バスに対する基本的CPUインターフェイスを図10-2に示す．

図10-3は8086システムの基本的インターフェイスを示す．8289バス・アービタは，バス・コントロールの移動に必要なプライオリティの判定とプロトコルのロジックを有する．このロジックはMultibusについて第9章に述べられている．8288バス・コントローラはCPUに対するシステム・バス・コントロール信号を備えており，システム・バスとのCPU

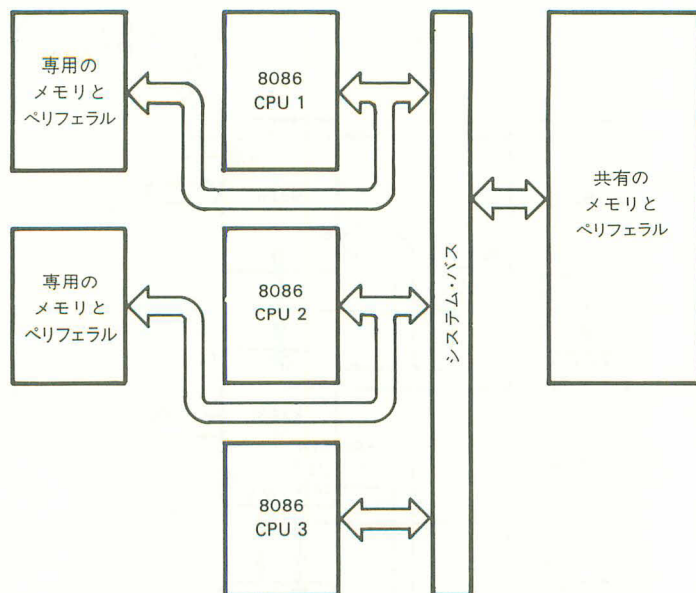



図10-2　マルチプロセッサ構成

のアドレスとデータのバス・インターフェイスも配慮している．8283と8287は共有システム・バスへのＣＰＵのインターフェイスを実現する．8288と8289はまた，システム・バス上の共有資源またはＣＰＵのローカル・バス上の固有資源への移動を管理する．

図10-3では，すべての資源，したがって，すべてのバス伝送はシステム・バスによって管理されている．バス・サイクル開始のために，8086は，パッシブ状態（$\overline{\text{S2}}$，$\overline{\text{S1}}$，$\overline{\text{S0}}$＝１，１，１）からアクティブ状態の１つにステータス・ラインを駆動する．8288と8289とＣＰＵは共通クロックＣＬＫによって同期的に動作し，ステータス・ラインをモニタすることによってバス・サイクル・リクエストを検出する．8288は，ローカルの多重化バスから8283へのアドレスのストローブのためにＡＬＥを発行する．8289がシステム・バスのコントロールを持たない（$\overline{\text{BUSY}}$をローに駆動していない）ならば，コントロールを得るためにバス・リクエスト（$\overline{\text{BREQ}}$）と共通バス・リクエスト（$\overline{\text{CBRQ}}$）を発行する．このプロトコルはMultibusのシステム・バスについて述べたものと同一である．

8289は，バス・コントロールを得るまで$\overline{\text{AEN}}$をインアクティブ（ハイ）状態に維持する．この動作は，8283のシステム・バスのアドレス駆動と，8288のバス・コマンドの駆動あるいは（ＤＥＮで）データ・トランシーバをイネーブルすることを防止する．8283は，$\overline{\text{OE}}$入力の状態とは無関係に，ＡＬＥの間にアドレスを捕える．$\overline{\text{AEN}}$のインアクティブ状態はまた，システムからの8284ＲＤＹ入力をディスエーブルとするのにも用いられる．ＲＤＹ入力のディスエーブルにより，このインターフェイスがバスのコントロールを獲得してバス・サイクルを終了するまで（ウェート状態の挿入によって）強制的にＣＰＵをウエートさせる．一度8289がバスのプライオリティを獲得してバスが動作中でなければ，$\overline{\text{AE}}$

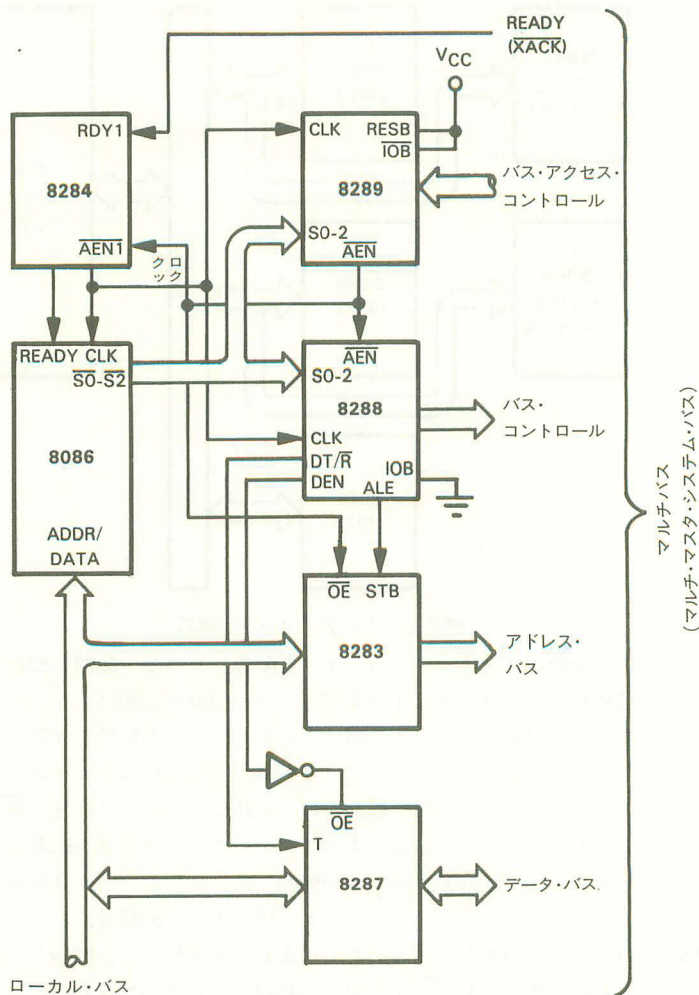



図10-3　ローカル資源のないＣＰＵ

$\overline{\text{N}}$をイネーブルとして他のバス・マスタに対して$\overline{\text{BUSY}}$を主張する．$\overline{\text{AEN}}$は直ちにバス上のアドレスをイネーブルにして，8288がデータ・バスのトランシーバ（8287）をイネーブルとすることを可能にする．

システムにおけるアドレス・セットアップとチップ・セレクト・デコードのタイムを可能とするために，105から275ns後まで8288はコマンドをイネーブルとしない．8284とＣＰＵが$\overline{\text{AEN}}$のイネーブルとなった直後にレディを検出してバス・サイクルを早まって終結することを防止するために，システムからのＲＤＹ入力は通常インアクティブで，8288によってコマンドが発行されるまでディスエーブルでなければならない．これを満たすために，伝送に関係して選択された素子は一般に，伝送が終了するまでＲＤＹ（Multibusのシ

ステム・バスの定義では$\overline{\mathrm{XACK}}$）を返さない．伝送を終了する時間は素子に依存しているので，各素子はコマンドの選択からＲＤＹ（または$\overline{\mathrm{XACK}}$）までに適当な遅延を備えている．

レディの検出後，ＣＰＵはステータス・ラインをパッシブ状態に戻す．この動作は，8288がコマンドを落としてトランシーバをディスエーブルにすることによって行なうバス・サイクルの8288の決定を可能にする．より高いプライオリティのバス・マスタが（バス・プライオリティ$\overline{\mathrm{BPRN}}$を失なうことによって）バス・サイクルの間にこのバス・マスタを強制的にバスから切り離そうとするならば，8289はステータス（$\overline{\mathrm{S2}}$，$\overline{\mathrm{S1}}$，$\overline{\mathrm{S0}}$）がパッシブ状態に戻るまで$\overline{\mathrm{BUSY}}$をアクティブ（ロー）に保つことによってバス・コントロールを維持する．8289がバスのコントロールを持てば，ＣＰＵによって開始されるバス・サイクルが直ちに実行されるのを可能にするために，$\overline{\mathrm{AEN}}$をアクティブに維持する．このような状況でのタイミングと信号シーケンスは，シングルＣＰＵのマキシマム・モード・システムと同一である．8289によってサポートされるバス放棄条件の十分な補足はこの章の後で述べる．

以前の解説では，すべてのメモリと I/O がマルチマスタ・システム・バスに接続されているシングルＣＰＵを表現している．バス・サイクルに利用可能なクロック・サイクル数を考慮すると，バス・サイクルにつき４クロックを仮定して，実行されるバス・サイクルの数を計算すると，応用が計算の限界かどうかに依存して，シングルの8086ＣＰＵは利用可能なバス帯域幅の50％と80％の間を利用できることが明らかになる．２つの8086が共有バスに所属しており，両プロセッサに対するすべてのメモリと I/O がこの共有バスに接続されていれば，各ＣＰＵについてのスループットは，次式から37％にまで悪化する．

$$\frac{80-50}{80}=37\%$$

この悪化は，共有バスへのアクセスに対する競合の直接の結果である．この計算では，各ＣＰＵが利用可能な全体のバス・サイクルの80％を利用するが，他のＣＰＵと１つのバスを共有しているので，利用可能なバス・サイクルの50％だけを与えられると仮定している．より多くのＣＰＵがシステムに付加されれば，それだけすべてのＣＰＵに悪化が起きる．マルチプロセッサ・システムから評価しうる利益を得るためには，高度な同時処理が存在しなければならない．したがって，共有バスへのアクセスの競合によるディレイは最小でなければならない．

この問題に対する１つの解として，8086ファミリーは各ＣＰＵに対して固有と共有の資源が定義されるのを可能にしている．固有の資源はメモリや I/O を含み，これはＣＰＵの固有またはレジデント・バスに接続され，割り当てられたＣＰＵによってだけアクセス可能である．それと対照に，マルチマスタ・システム・バスに直接に接続され，１つ以上のマスタによってアクセス可能なメモリと I/O から共有資源は構成される．

資源分割の主要な方法，アドレス・マッピングとメモリ対 I/O は，8288と8289によってサポートされる．**図10-4(a)**から**10-4(c)**までは，この方法から導かれる変形を示す．これらすべての例における共通の要素に，第２の組のアドレス・ラッチを含むことと8284の２つ

図10-4(a)　ローカルＲＯＭ／ＥＰＲＯＭを有する8086

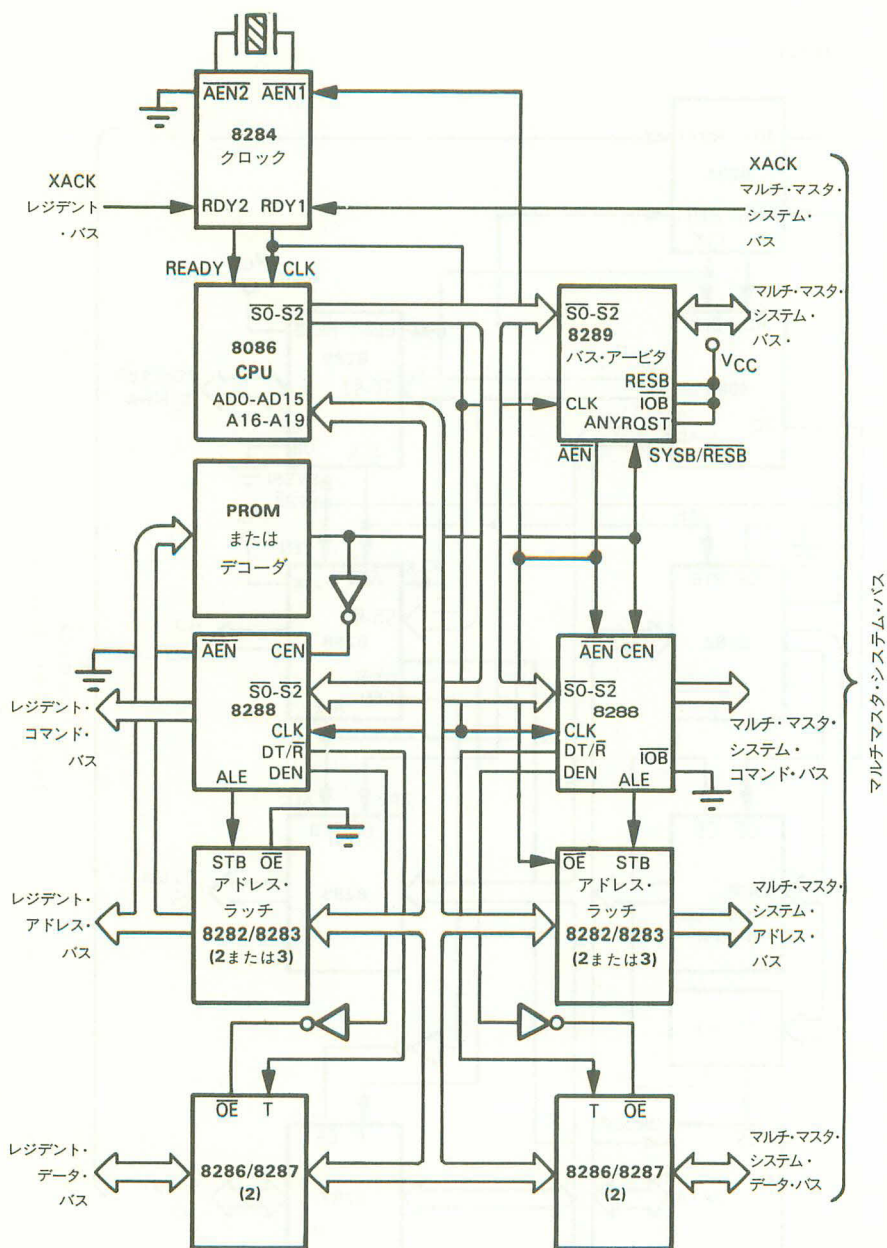
図10-4(b)　ローカルＲＯＭ／ＥＰＲＯＭとI/O を有する8086

* 別の8289アービタを加えてその$\overline{\text{AEN}}$を現在$\overline{\text{AEN}}$が接地されている8289に接続することによって，プロセッサは2つの マルチ・マスタ・バスにアクセスすることができる．

図10-4(c)　ローカルＲＡＭ／ＲＯＭ／ＥＰＲＯＭ／Ｉ／Ｏを有する8086

のＲＤＹ入力の使用がある．図10-4(a)から10-4(c)に示す構成の主要な差異は，固有資源のサポートに必要な付加的要素の数とサポートされる固有資源のタイプにある．

図10-4(a)はＣＰＵのローカル・バス上にＲＯＭあるいはＥＰＲＯＭの存在を認め，付加的アドレス・ラッチと可能性のあるアドレス・デコード・ラッチだけを必要とする．8289はレジデント・バス・モードに切り替えられ（ＲＥＳＢをＶccに)，これは8289にこのＣＰＵがレジデント・メモリをサポートすることを通知する．8288の切り替えによる選択は，図10-3に示す構成と異ならない．バス・サイクルの間に，8288はＡＬＥを発行し，8283と8282にアドレスをラッチする．レジデント・メモリに割り当てられたアドレス領域はデコードされて，このバス・サイクルがレジデントあるいはシステムの資源を用いるかの判断に利用される．バス・サイクルがシステム・バスを利用するならば，デコードの結果は，8289に対してＳＹＳＢを，8288に対してＣＥＮを示し，関連のある$\overline{\text{ＡＥＮ}}$によってローカル・バスのＲＤＹをディスエーブルにするために，ハイでなければならない．

8289がＲＥＳＢモードのとき，8289に対するＳＹＳＢ／$\overline{\text{ＲＥＳＢ}}$入力は，8289がシステム・バスを要求するか解放するかを決定する．ＳＹＳＢ／$\overline{\text{ＲＥＳＢ}}$がハイならば，8289はシステム・バスを要求し，ＳＹＳＢ／$\overline{\text{ＲＥＳＢ}}$がローならば，8289はバスを解放する．8288のＣＥＮ入力は，コントロール出力がレジデント・バス・アクセスの間にアクティブ状態に駆動されるのを防止する．この信号は，8289がバスのコントロールを有するならば，コントロール信号がシステムに対して発行されるのを防ぐ．8289がバスのコントロールを有する間は，8288はコントロール出力をアクティブ状態に維持しなければならないので，入力によって8288はコントロール出力をフロートにはしない．8288は常にイネーブル（$\overline{\text{ＯＥ}}$はローに接続）になっていることに注意．これにより，各バス・サイクルのアドレスの適当なシステム・バスまたはレジデント・バスの選択への早いデコードが可能となる．

8288にラッチされる安定なアドレスはまた，固有のＲＯＭまたはＥＰＲＯＭにチップ・セレクトとアドレスを供給する．8086からのリード信号（$\overline{\text{ＲＤ}}$）は，選択された固有素子の固有多重化バスへのデータの駆動をイネーブルとするために用いられる．固有資源が選択されると，8284の$\overline{\text{ＡＥＮ２}}$入力は，ローカルのＲＤＹ２のＣＰＵに対するレディの生成を可能とするために，イネーブルとなる．8288はシステム・バスのコントロール信号を駆動しないので，8289がバス・コントロールを有して$\overline{\text{ＡＥＮ１}}$がアクティブであっても，ＲＤＹ１（または$\overline{\text{ＸＡＣＫ}}$）を受け付けることはできない．ローカル資源がローカル多重化バスからのバッファを必要とするならば，ローカルのチップ・セレクトと$\overline{\text{ＲＤ}}$はバッファをイネーブルとするために用いられる必要がある．I/Oアドレス領域がローカル・メモリ・アドレス領域とオーバラップするならば，I/Oアドレスがレジデント・バス・アクセスにデコードされるのを防止するために，$\overline{\text{Ｓ２}}$はラッチされてローカル・アドレス・デコードに含まれなければならない．

I/Oコマンドまたはライト・コマンドはローカル・バスに利用できないので，この方法はメモリ領域のローカルなＲＯＭとＥＰＲＯＭに対してだけ動作する．ただし，オブジェクト・コードのアクセスは一般にバス・サイクルの50％から70％を利用することに注意．したがって，ＣＰＵによって実行されるオブジェクト・コードの大部分がローカルのＲＯ

ＭあるいはＥＰＲＯＭに存在すれば，ＣＰＵあたりのシステム・バス利用度は30%以下にまで低下する．すべてのＲＡＭと I/O は共有システム・バスに位置せねばならず，したがって競合の可能性を持つ．

図10-4(b)は，図10-4(a)に示す前の構成にローカルの I/O を付加している．これは，8288と8289を共にＩＯＢ（I/Oバス）モードの動作に切り替えることによって達せられる．このモードでは，すべての I/O はローカル・バス上に位置すると仮定され，したがって I/Oはシステム・バスに対するアクセスを要求しない．しかし，ＩＯＢの選択が用いられるとき，メモリ・マップでなければ，ＣＰＵはシステム・バス上の I/O にアクセスすることができない．各 I/O のバス伝送に対して，8289はバスのコントロールを獲得しようとはせず，また現在コントロールを有してもバスを解放する．$\overline{\mathrm{AEN}}$入力の状態とは無関係に，8288は I/O のコントロール出力を駆動する．このモードでは，8288の I/O コントロール出力は，システム・バスのコントロール・ラインに接続されてはならない．これは，ローカルの I/O 素子に対するコントロールを供給するためにだけ用いられる．図10-4(a)に対して，この拡張をサポートするために付加的な回路は必要としない．

8284の$\overline{\mathrm{AEN2}}$入力は，ローカル・メモリあるいはローカル I/O に対するチップ・セレクトによってイネーブルとなる．8288のＣＥＮ入力は，I/Oコントロール出力がロー（アクティブ）に駆動されるのを可能とするために，I/Oサイクルの間はハイでなければならない．I/O素子に対するデータ・ラインがローカル多重化バスとの間にバッファが必要ならば，8288からのＤＴ／$\overline{\mathrm{R}}$信号はバッファに対する方向コントロール信号として用いることができる．バッファをイネーブルとするために，8288は別のペリフェラル・データ・イネーブル・コントロール（$\overline{\mathrm{PDEN}}$）を備えている．この信号は，8288が I/O バス（ＩＯＢ）モードに切り替えられているときにだけ利用できる．$\overline{\mathrm{PDEN}}$は I/O 伝送に対してイネーブルとなり，ＤＥＮはシステム・バスによるデータ伝送に対してだけイネーブルとなる．

別の構成に対する次の拡張を図10-4(c)に示す．このケースに対しては，第２の8288とデータ・バス・トランシーバが付加される．この構成は，完全にアドレス・マッピングに基づいているので，最も融通性があり，共有とローカルのＲＯＭ／ＥＰＲＯＭ，ＲＡＭとＩ／Ｏへのアクセスが可能である．ＩＯＢモードはどちらの8288にも用いられていない．ローカル・バスはマルチマスタ・バスではないので，第２の8289は必要とされない．アドレス・マップにおいてＰＲＯＭあるいはデコーダ・ソースがメモリと I/O の参照に対して存在する．結果のＳＹＳＢ／$\overline{\mathrm{RESB}}$信号は，レジデント・バスの8288ＣＥＮ入力に対して適当なイネーブルの極性を供給するために反転される．8288は，レジデント・データ・バスに対してアドレスのラッチと完全なデータ・バスのトランシーバ・コントロールを，そしてメモリと I/O に対して読み込みと書き込みのコントロールの完全な集合を供給する．マルチマスタ・システム・バス・インターフェイスは，図10-4(a)の構成に対して述べたのと同様に機能する．

図10-4(c)の構成は，固定されたプログラムと定数のためのレジデントＲＯＭ／ＥＰＲＯＭ，スタックとローカル変数のためのレジデントＲＡＭ，そしてこのＣＰＵだけがアクセ

スを必要とするレジデント I/O を，CPUがサポートすることを可能にする．結果は，共有のメモリまたは I/O によってコントロールとデータを伝えるモジュールの完備した処理モジュールとなる．これにより，CPUあたりの代表的なシステム・バスの使用が25%以下にまで減少したシステムで，高度な同時処理が可能となる．

CPUに対する多重バスの概念の最後の拡張は，レジデント・バスではなくて第2のマルチマスタ・バスの実現である．これはフォールト・トレラント・システムに有用であり，また，複数プロセッサ間コミュニケーション・チャンネルに基づく性能の増強を可能にする．この場合，システム・バスにアクセスしようとするCPUは，プライマリ・バスを利用しようとするが，これが利用できないならば，セカンダリ・バスにアクセスする．

10.3　8289のバス・アクセスとレリーズ・オプション

RESBとIOBのモードに加えて，8289はマルチマスタ・システム・バスの利用を最適にするいくつかの他のオプションを有している．この付加的機能は，バスの要求よりも解放に影響を与える．バス・リクエストの動作に関して，1つのコメントを加える必要がある．RESBモードを除いて，T2のローからハイへのCPUクロックの遷移に続く第2のハイからローへのクロック遷移（BCLK）で，8289はバス・リクエストを発行する．RESBモードに対する1つのCPUクロックのディレイは，安定なSYSB/$\overline{\mathrm{RESB}}$信号発生のためのアドレス・デコードの時間を与える．

バス・コントロールの解放についての付加的コントロールのために，8289は$\overline{\mathrm{LOCK}}$，ANYRQST，そして$\overline{\mathrm{CRQLCK}}$の入力を備えている．一般に，8289はプライオリティを失なった（$\overline{\mathrm{BPRN}}$がハイになった）ときに,現在のサイクルの終わりまたは(バス・サイクルが進行中でなければ）ただちにバスのコントロールを放棄する．$\overline{\mathrm{LOCK}}$により，バス・プライオリティとは無関係に，バス・コントロールを維持する（$\overline{\mathrm{BUSY}}$をローに駆動し続ける）ことができる．$\overline{\mathrm{LOCK}}$先行の命令実行の間とインタラプト・アクノリッジ・シーケンスの間にCPUがバス・コントロールを維持することを保証するために，8289の$\overline{\mathrm{LOCK}}$入力はCPUの$\overline{\mathrm{LOCK}}$出力によって駆動されなければならない．$\overline{\mathrm{LOCK}}$信号により，ロックされた交換のような動作が，ノンリエントラント共有資源の基本的セマフォ・コントロール（クリティカル・コード・セクションとしても知られている）に対してプリミティブとして動作することが可能になる．

ANYRQSTは，コモン・バス・リクエスト（$\overline{\mathrm{CBRQ}}$）がアクティブならば，現在のバス・サイクルの終わりあるいは（バス・サイクルが進行中でなければ）直ちに強制的に8289にバスを解放させる，切り替えによる選択である．

ANYRQSTの主張は，強制的により低いプライオリティのバス・マスタへ8289にバスを解放させる．このケースに対するバスの解放には，8289がそのバス・リクエストを落とし（$\overline{\mathrm{BREQ}}$をハイに戻し），$\overline{\mathrm{BPRO}}$（バス・プライオリティ・アウト）をイネーブルとする必要がある．（$\overline{\mathrm{BREQ}}$は並列プライオリティ解決回路に用いられ，一方$\overline{\mathrm{BPRO}}$は次に低いプライオリティの$\overline{\mathrm{BPRN}}$に対する$\overline{\mathrm{BPRO}}$のディジィ・チェーンによる直列プラ

イオリティ判定方法をサポートするために供給されることに注意．２つの方法のうち１つだけが単一のシステムに適用される．

次いで8289は，ＢＵＳＹを解放することによって１つのバス・クロック期間後にバスを解放する．より低いプライオリティの8289が$\overline{BPRN}$を受け取ることを可能とするために，8289は$\overline{BREQ}$と$\overline{BPRO}$を解放しなければならない．他の8289をバスから強制的に切り離すより高いプライオリティのマスタの通常のケースに対して，より高いプライオリティのマスタはプライオリティの機構から$\overline{BPRN}$を受け取り，より低いプライオリティのマスタがその$\overline{BPRO}$または$\overline{BREQ}$を解放することを必要としない．

ANYRQSTがハイに切り替えられて$\overline{CBRQ}$がローに接続されていれば，8289はバス・サイクルの終了後にバスを解放する．これはバス伝送ロジックに高度なオーバーヘッドを課するが，めったにシステム・バスを使用しないか，あるいは非常に低い帯域幅で用いるマスタに対しては有用である．

ＡＮＹＲＱＳＴがハイでなければ，ＣＰＵのローカル・バスがアイドルのとき（$\overline{S2}$，$\overline{S1}$，$\overline{S0}$＝１，１，１）に限り，$\overline{CBRQ}$要求のより低いプライオリティの素子に対して8289はバスを解放する．$\overline{CBRQ}$を無効にしてより低いプライオリティのマスタに対するバス解放を行なわないために，$\overline{CRQLCK}$（コモン・バス・リクエスト・ロック）がある．この入力をローにすることによって，入力としての$\overline{CBRQ}$を有効にディスエーブルにできる．同様の効果は$\overline{CBRQ}$をハイに接続することによって達成できるが，$\overline{CRQLCK}$によってスタティックな機能をプログラム可能な選択とすることができる．

$\overline{CBRQ}$は双方向で，8289はバスを獲得するためにローに駆動し，バスを有するときはモニタすることに注意．したがって，バスを獲得するためにバスにまだ$\overline{CBRQ}$を利用させている間に，8289が$\overline{CBRQ}$を無視することを$\overline{CRQLCK}$は認めている．

8289にバスを解放させる他の特殊な条件には，8289が I/O バス（ＩＯＢ）モードに切り替えられている状態での I/O サイクル，8289がＲＥＳＢモードとなっているレジデント・バス・サイクル，そして常にＣＰＵがＨＡＬＴ命令の実行によってホルト状態に入るときがあげられる．ホルト状態に入ることは，ステータス・ラインのホルト・ステータス（$\overline{S2}$，$\overline{S1}$，$\overline{S0}$＝０，１，１）によって8289に示される．

付録A

8086命令セット一覧

―アルファベット順―

命　令		オブジェクト・コード	バイト	クロック
AAA		37	1	4
AAD		D5 0A	2	60
AAM		D4 0A	2	83
AAS		3F	1	4
ADC	ac,data	0001010w kk [jj]	2 または 3	4
ADC	mem/reg$_1$,data	100000sw mod 010 r/m [DISP] [DISP] kk [jj]	3, 4, 5 または 6	reg: 4 mem: 17 + EA
ADC	mem/reg$_1$,mem/reg$_2$	000100dw mod rrr r/m [DISP] [DISP]	2,3 または 4	reg to reg: 3 mem to reg: 9 + EA reg to mem: 16 + EA
ADD	ac,data	0000010w kk [jj]	2 または 3	4
ADD	mem/reg,data	100000sw mod 000 r/m [DISP] [DISP] kk [jj]	3, 4, 5 または 6	reg: 4 mem: 17 + EA
ADD	mem/reg$_1$,mem/reg$_2$	000000dw mod rrr r/m [DISP] [DISP]	2,3 または 4	reg to reg: 3 mem to reg: 9 + EA reg to mem: 16 + EA
AND	ac,data	0010010w kk [jj]	2 または 3	4
AND	mem/reg,data	1000000w mod 100 r/m [DISP] [DISP] kk [jj]	3, 4, 5 または 6	reg: 4 mem:17 + EA
AND	mem/reg$_1$,mem/reg$_2$	001000dw mod rrr r/m [DISP] [DISP]	2,3 または 4	reg to reg: 3 mem to reg: 9 + EA reg to mem: 16 + EA
CALL	addr	9A kk jj hh gg	5	28
CALL	disp16	E8 kk jj	3	19
CALL	mem	FF mod 011 r/m [DISP] [DISP]	2,3 または 4	32-bit mem ポインタ: 37 + EA
CALL	mem/reg	FF mod 010 r/m [DISP] [DISP]	2,3 または 4	16-bit reg ポインタ: 16 16-bit mem ポインタ: 21 + EA

命　令		オブジェクト・コード	バイト	クロック
CBW		98	1	2
CLC		F8	1	2
CLD		FC	1	2
CLI		FA	1	2
CMC		F5	1	2
CMP	ac,data	0011110w kk [jj]	2 または 3	4
CMP	mem/reg,data	100000sw mod 111 r/m [DISP] [DISP] kk [jj]	3, 4, 5 または 6	reg: 4 mem: 10 + EA
CMP	mem/reg1,mem/reg2	001110dw mod rrr r/m [DISP] [DISP]	2,3 または 4	reg to reg: 3 mem to reg: 9 + EA reg to mem: 9 + EA
CMPS		1010011w	1	22 9 + 22/繰返し*
CWD		99	1	5
DAA		27	1	4
DAS		2F	1	4
DEC	mem/reg	1111111w mod 001 r/m [DISP] [DISP]	2,3 または 4	reg: 3 mem: 15 + EA
DEC	16-bit reg	01001rrr	1	2
DIV	mem/reg	1111011w mod 110 r/m [DISP] [DISP]	2,3 または 4	8-bit reg: 80 → 90 16-bit reg: 144 → 162 8-bit mem: (86 → 96) + EA 16-bit mem: (150 → 168) + EA
ESC	mem/reg	11011xxx mod xxx r/m [DISP] [DISP]	2,3 または 4	mem: 8 + EA reg: 2
HLT		F4	1	2
IDIV	mem/reg	1111011w mod 111 r/m [DISP] [DISP]	2,3 または 4	8-bit reg: 101 → 112 16-bit reg: 165 → 184 8-bit mem: (107 → 118) + EA 16-bit mem: (171 → 190) + EA
IMUL	mem/reg	1111011w mod 101 r/m [DISP] [DISP]	2,3 または 4	8-bit reg: 80 → 98 16-bit reg: 128 → 154 8-bit mem: (86 → 104) + EA 16-bit mem: (134 → 160) + EA
IN	ac, DX	1110110w	1	8
IN	ac, port	1110010w	2	10

* REPプレフィックスが先行したとき

命　令		オブジェクト・コード	バイト	クロック
INC	mem/reg	1111111w mod 000 r/m [DISP] [DISP]	2,3 または 4	reg: 3 mem: 15 + EA
INC	16-bit reg	01000rrr	1	2
INT		11001100* 11001101 type	1 2	52 51
INTO		CE	1	interrupt: 53 no interrupt: 4
IRET		CF	1	32
JA JNBE	disp	77 disp	2	4/No Branch 16/Branch
JAE JNB	disp	73 disp	2	4/No Branch 16/Branch
JB JNAE	disp	72 disp	2	4/No Branch 8/Branch
JBE JNA	disp	76 disp	2	4/No Branch 16/Branch
JCXZ	disp	E3 disp	2	6/No Branch 18/Branch
JE JZ	disp	74 disp	2	4/No Branch 16/Branch
JG JNLE	disp	7F disp	2	4/No Branch 16/Branch
JGE JNL	disp	7D disp	2	4/No Branch 16/Branch
JL JNGE	disp	7C disp	2	4/No Branch 16/Branch
JLE JNG	disp	7E disp	2	4/No Branch 16/Branch
JMP	addr	EA kk jj hh gg	5	15
JMP	disp	EB disp	2	15
JMP	disp16	E9 kk jj	3	15
JMP	mem	FF mod 101 r/m [DISP] [DISP]	2,3 または 4	mem ptr 32: 24 + EA
JMP	mem/reg	FF mod 100 rr/m [DISP] [DISP]	2,3 または 4	reg ptr 16: 11 mem ptr 16: 16 + EA
JNE JNZ	disp	75 disp	2	4/No Branch 16/Branch
JNO	disp	71 disp	2	4/No Branch 16/Branch
JNP JPO	disp	7B disp	2	4/No Branch 16/Branch
JNS	disp	79 disp	2	4/No Branch 16/Branch
JO	disp	70 disp	2	4/No Branch 16/Branch

* type=3を仮定

命令		オブジェクト・コード	バイト	クロック
JP JPE	disp	7A disp	2	4/No Branch 16/Branch
JS	disp	78 disp	2	4/No Branch 16/Branch
LAHF		9F	1	4
LDS	reg,mem	C5 mod rrr r/m [DISP] [DISP]	2,3 または 4	16 + EA
LEA	reg,mem	8D mod rrr r/m [DISP] [DISP]	2,3 または 4	2 + EA
LES	reg,mem	C4 mod rrr r/m [DISP] [DISP]	2,3 または 4	16 + EA
LOCK		F0	1	2
LODS		1010110w	1	12 9 + 13/繰返し*
LOOP	disp	E2 disp	2	5/No Branch 17/Branch
LOOPE LOOPZ	disp	E1 disp	2	6/No Branch 18/Branch
LOOPNE LOOPNZ	disp	E0 disp	2	5/No Branch 19/Branch
MOV	$mem/reg_1,mem/reg_2$	100010dw mod rrr r/m [DISP] [DISP]	2,3 または 4	reg to reg: 2 reg to mem: 8 + EA mem to reg: 9 + EA
MOV	reg,data	1011wrrr kk [jj]	2 または 3	4
MOV	ac,mem	1010000w kk jj	3	10
MOV	mem,ac	1010001w kk jj	3	10
MOV	segreg,mem/reg	8E mod 0rr r/m [DISP] [DISP]	2,3 または 4	reg to reg: 2 mem to reg: 8 + EA
MOV	mem/reg,segreg	8C mod 0rr r/m [DISP] [DISP]	2,3 または 4	reg to reg: 2 reg to mem: 9 + EA
MOV	mem/reg,data	1100011w mod 000 r/m [DISP] [DISP] kk [jj]	3, 4, 5 または 6	reg/mem: 10 + EA
MOVS		1010010w	1	18 9 + 17/繰返し*

* REPプレフィックスが先行したとき

命　令		オブジェクト・コード	バイト	クロック
MUL	mem/reg	1111011w mod 100 r/m [DISP] [DISP]	2,3 または 4	8-bit reg: 70 → 77 16-bit reg: 118 → 133 8-bit mem: (76 → 83) + EA 16-bit mem: (124 → 139) + EA
NEG	mem/reg	1111011w mod 011 r/m [DISP] [DISP]	2,3 または 4	reg: 3 mem: 16 + EA
NOP		90	1	3
NOT	mem/reg	1111011w mod 010 r/m [DISP] [DISP]	2,3 または 4	reg: 3 mem: 16 + EA
OR	ac,data	0000110w kk [jj]	2 または 3	4
OR	mem/reg,data	1000000w mod 001 r/m [DISP] [DISP] kk [jj]	3, 4, 5 または 6	reg: 4 mem: 17 + EA
OR	mem/reg1,mem/reg2	000010dw mod rrr r/m [DISP] [DISP] kk [jj]	3,4,5 または 6	reg to reg: 3 mem to reg: 9 + EA reg to mem: 16 + EA
OUT	DX,ac	1110111w	1	8
OUT	port,ac	1110011w yy	2	10
POP	mem/reg	8F mod 000 r/m [DISP] [DISP]	2,3 または 4	reg: 8 mem: 17 + EA
POP	reg	01011rrr	1	8
POP	segreg	000ss111	1	8
POPF		9D	1	8
PUSH	mem/reg	FF mod 110 r/m [DISP] [DISP]	2,3 または 4	reg: 11 mem: 16 + EA
PUSH	reg	01010rrr	1	10
PUSH	segreg	000ss110	1	10
PUSHF•		9C	1	10
RCL	mem/reg,count	110100cw mod 010 r/m [DISP] [DISP]	2,3 または 4	count = 1 reg: 2 mem:15 + EA count = [CL] reg: 8 + (4 * N) mem: 20 + EA + (4 * N)

NはCL内の値

命　令		オブジェクト・コード	バイト	クロック
RCR	mem/reg,count	110100cw mod 011 r/m [DISP] [DISP]	2,3 または 4	count = 1 reg: 2 mem:15 + EA count = [CL] reg: 8 + (4 * N) mem: 20 + EA + (4 * N)
REP	/REPE/REPNE	1111001z	1	2
RET	(セグメント間)	CB	1	24
RET	(セグメント内)	C3	1	16
RET	disp16(セグメント間)	CA kk jj	3	23
RET	disp16(セグメント内)	C2 kk jj	3	20
ROL	mem/reg,count	110100cw mod 000 r/m [DISP] [DISP]	2,3 または 4	count = 1 reg: 2 mem:15 + EA count = [CL] reg: 8 + (4 * N) mem: 20 + EA + (4 * N)
ROR	mem/reg,count	110100cw mod 001 r/m [DISP] [DISP]	2,3 または 4	count = 1 reg: 2 mem:15 + EA count = [CL] reg: 8 + (4 * N) mem: 20 + EA + (4 * N)
SAHF		9E	1	4
SAR	mem/reg,count	110100cw mod 111 r/m [DISP] [DISP]	2,3 または 4	count = 1 reg: 2 mem:15 + EA count = [CL] reg: 8 + (4 * N) mem: 20 + EA + (4 * N)
SBB	ac,data	0001110w kk [jj]	2 または 3	4
SBB	mem/reg,data	100000sw mod 011 r/m [DISP] [DISP] kk [jj]	3, 4, 5 または 6	reg: 4 mem: 17 + EA
SBB	mem/reg$_1$,mem/reg$_2$	000110dw mod rrr r/m [DISP] [DISP]	2,3 または 4	reg from reg: 3 mem from reg: 9 + EA reg from mem: 16 + EA
SCAS		1010111w	1	15 9 + 15/繰返し*
SEG	segreg	001ss110	1	2
SHL SAL	mem/reg,count	110100cw mod 100 r/m [DISP] [DISP]	2,3 または 4	count = 1 reg: 2 mem:15 + EA count = [CL] reg: 8 + (4 * N) mem: 20 + EA + (4 * N)

* REPプレフィックスが先行したとき
NはCL内の値

命令		オブジェクト・コード	バイト	クロック
SHR	mem/reg,count	110100cw mod 101 r/m [DISP] [DISP]	2,3 または 4	count = 1 reg: 2 mem:15 + EA count = [CL] reg: 8 + (4 * N) mem: 20 + EA + (4 * N)
STC		F9	1	2
STD		FD	1	2
STI		FB	1	2
STOS		1010101w	1	11 9 + 10/繰返し*
SUB	ac,data	0010110w kk [jj]	2 または 3	4
SUB	mem/reg,data	100000sw mod 101 r/m [DISP] [DISP] kk [jj]	3, 4, 5 または 6	reg: 4 mem: 17 + EA
SUB	$mem/reg_1,mem/reg_2$	001010dw mod rrr r/m [DISP] [DISP]	2,3 または 4	reg from reg: 3 mem from reg: 9 + EA reg from mem: 16 + EA
TEST	ac,data	1010100w kk [jj]	2 または 3	4
TEST	mem/reg,data	1111011w mod 000 r/m [DISP] [DISP] kk [jj]	3, 4, 5 または 6	reg: 5 mem: 11 + EA
TEST	reg,mem/reg	1000010w mod rrr r/m [DISP] [DISP]	2,3 または 4	reg with reg: 3 reg with mem: 9 + EA
WAIT		9B	1	3(最小)+ 5n
XCHG	reg,ac	10010rrr	1	3
XCHG	reg,mem/reg	1000011w mod rrr r/m [DISP] [DISP]	2,3 または 4	reg with reg: 4 reg with mem: 17 + EA
XLAT		D7	1	11
XOR	ac,data	0011010w kk [jj]	2 または 3	4
XOR	mem/reg,data	1000000w mod 110 r/m [DISP] [DISP] kk [jj]	3, 4, 5 または 6	reg: 4 mem: 17 + EA
XOR	$mem/reg_1,mem/reg_2$	001100dw mod rrr r/m [DISP] [DISP]	2,3 または 4	reg with reg: 3 mem with reg: 9 + EA reg with mem: 16 + EA

* REPプレフィックスが先行したとき
n は，$\overline{\text{TEST}}$ インプットのサンプル当たりのクロック

付録B

8086命令セット一覧

—オブジェクト・コード数値上昇順—

オブジェクト・コード			ニーモニック
バイト＃0	バイト＃1	後続バイト	
00	mod reg r/m	[disp][disp]	ADD mem/reg,reg（バイト）
01	mod reg r/m	[disp][disp]	ADD mem/reg,reg（ワード）
02	mod reg r/m	[disp][disp]	ADD reg, mem/reg（バイト）
03	mod reg r/m	[disp][disp]	ADD reg, mem/reg（ワード）
04	kk		ADD AL,kk
05	kk	jj	ADD AX, jjkk
06			PUSH ES
07			POP ES
08	mod reg r/m	[disp][disp]	OR mem/reg,reg（バイト）
09	mod reg r/m	[disp][disp]	OR mem/reg,reg（ワード）
0A	mod reg r/m	[disp][disp]	OR reg,mem/reg（バイト）
0B	mod reg r/m	[disp][disp]	OR reg,mem/reg（ワード）
0C	kk		OR AL,kk
0D	kk	jj	OR AL,jjkk
0E			PUSH CS
0F			未使用
10	mod reg r/m	[disp][disp]	ADC mem/reg,reg（バイト）
11	mod reg r/m	[disp][disp]	ADC mem/reg,reg（ワード）
12	mod reg r/m	[disp][disp]	ADC reg,mem/reg（バイト）
13	mod reg r/m	[disp][disp]	ADC reg,mem/reg（ワード）
14	kk		ADC AL,kk
15	kk	jj	ADC AX,jjkk
16			PUSH SS
17			POP SS
18	mod reg r/m	[disp][disp]	SBB mem/reg,reg（バイト）
19	mod reg r/m	[disp][disp]	SBB mem/reg,reg（ワード）
1A	mod reg r/m	[disp][disp]	SBB reg,mem/reg（バイト）
1B	mod reg r/m	[disp][disp]	SBB reg,mem/reg（ワード）
1C	kk		SBB AL,kk
1D	kk	jj	SBB AX,jjkk
1E			PUSH DS
1F			POP DS
20	mod reg r/m	[disp][disp]	AND mem/reg,reg（バイト）
21	mod reg r/m	[disp][disp]	AND mem/reg,reg（ワード）
22	mod reg r/m	[disp][disp]	AND reg,mem/reg（バイト）
23	mod reg r/m	[disp][disp]	AND reg,mem/reg（ワード）
24	kk		AND AL,kk
25	kk	jj	AND AX,jjkk
26			SEG ES
27			DAA
28	mod reg r/m	[disp][disp]	SUB mem/reg,reg（バイト）
29	mod reg r/m	[disp][disp]	SUB mem/reg,reg（ワード）
2A	mod reg r/m	[disp][disp]	SUB reg,mem/reg（バイト）
2B	mod reg r/m	[disp][disp]	SUB reg,mem/reg（ワード）
2C	kk		SUB AL,kk
2D	kk	jj	SUB AX,jjkk
2E			SEG CS
2F			DAS

オブジェクト・コード			ニーモニック
バイト＃0	バイト＃1	後続バイト	
30	mod reg r/m	[disp][disp]	XOR mem/reg,reg（バイト）
31	mod reg r/m	[disp][disp]	XOR mem/reg,reg（ワード）
32	mod reg r/m	[disp][disp]	XOR reg,mem/reg（バイト）
33	mod reg r/m	[disp][disp]	XOR reg,mem/reg（ワード）
34	kk		XOR AL,kk
35	kk	jj	XOR AX,jjkk
36			SEG SS
37			AAA
38	mod reg r/m	[disp][disp]	CMP mem/reg,reg（バイト）
39	mod reg r/m	[disp][disp]	CMP mem/reg,reg（ワード）
3A	mod reg r/m	[disp][disp]	CMP reg,mem/reg（バイト）
3B	mod reg r/m	[disp][disp]	CMP reg,mem/reg（ワード）
3C	kk		CMP AL,kk
3D	kk	jj	CMP AX,jjkk
3E			SEG DS
3F			AAS
40			INC AX
41			INC CX
42			INC DX
43			INC BX
44			INC SP
45			INC BP
46			INC SI
47			INC DI
48			DEC AX
49			DEC CX
4A			DEC DX
4B			DEC BX
4C			DEC SP
4D			DEC BP
4E			DEC SI
4F			DEC DI
50			PUSH AX
51			PUSH CX
52			PUSH DX
53			PUSH BX
54			PUSH SP
55			PUSH BP
56			PUSH SI
57			PUSH DI
58			POP AX
59			POP CX
5A			POP DX
5B			POP BX
5C			POP SP
5D			POP BP
5E			POP SI
5F			POP DI
60-6F			未使用

オブジェクト・コード			ニーモニック
バイト＃0	バイト＃1	後続バイト	
70	disp		JO disp
71	disp		JNO disp
72	disp		JB or JNAE or JC disp
73	disp		JNB or JAE or JNC disp
74	disp		JE or JZ disp
75	disp		JNE or JNZ disp
76	disp		JBE or JNA disp
77	disp		JNBE or JA disp
78	disp		JS disp
79	disp		JNS disp
7A	disp		JP or JPE disp
7B	disp		JNP or JPO disp
7C	disp		JL or JNGE disp
7D	disp		JNL or JGE disp
7E	disp		JLE or JNG disp
7F	disp		JNLE or JG disp
80	mod 000 r/m	[disp][disp] kk	ADD mem/reg,kk
80	mod 001 r/m	[disp][disp] kk	OR mem/reg,kk
80	mod 010 r/m	[disp][disp] kk	ADC mem/reg,kk
80	mod 011 r/m	[disp][disp] kk	SBB mem/reg,kk
80	mod 100 r/m	[disp][disp] kk	AND mem/reg,kk
80	mod 101 r/m	[disp][disp] kk	SUB mem/reg,kk
80	mod 110 r/m	[disp][disp] kk	XOR mem/reg, kk
80	mod 111 r/m	[disp][disp] kk	CMP mem/reg,kk
81	mod 000 r/m	[disp][disp] kkjj	ADD mem/reg,jjkk
81	mod 001 r/m	[disp][disp] kkjj	OR mem/reg,jjkk
81	mod 010 r/m	[disp][disp] kkjj	ADC mem/reg,jjkk
81	mod 011 r/m	[disp][disp] kkjj	SBB mem/reg,jjkk
81	mod 100 r/m	[disp][disp] kkjj	AND mem/reg,jjkk
81	mod 101 r/m	[disp][disp] kkjj	SUB mem/reg,jjkk
81	mod 110 r/m	[disp][disp] kkjj	XOR mem/reg,jjkk
81	mod 111 r/m	[disp][disp] kkjj	CMP mem/reg,jjkk
82	mod 000 r/m	[disp][disp] kk	ADD mem/reg,kk（バイト）
82	xx 001 xxx		未使用
82	mod 010 r/m	[disp][disp] kk	ADC mem/reg,kk（バイト）
82	mod 011 r/m	[disp][disp] kk	SBB mem/reg,kk（バイト）
82	xx 100 xxx		未使用
82	mod 101 r/m	[disp][disp] kk	SUB mem/reg,kk（バイト）
82	xx 110 xxx		未使用
82	mod 111 r/m	[disp][disp] kk	CMP mem/reg,kk（バイト）
83	mod 000 r/m	[disp][disp] kk	ADD mem/reg,jjkk（ワードに符号拡張）
83	xx 001 xxx		未使用
83	mod 010 r/m	[disp][disp] kk	ADC mem/reg,jjkk（ワードに符号拡張）
83	mod 011 r/m	[disp][disp] kk	SBB mem/reg,jjkk（ワードに符号拡張）
83	xx 100 r/m		未使用
83	mod 101 r/m	[disp][disp] kk	SUB mem/reg,jjkk（ワードに符号拡張）
83	xx 110 xxx		未使用
83	mod 111 r/m	[disp][disp] kk	CMP mem/reg,jjkk（ワードに符号拡張）
84	mod reg r/m	[disp][disp]	TEST mem/reg,reg（バイト）
85	mod reg r/m	[disp][disp]	TEST mem/reg,reg（ワード）
86	mod reg r/m	[disp][disp]	XCHG reg,mem/reg（バイト）
87	mod reg r/m	[disp][disp]	XCHG reg,mem/reg（ワード）
88	mod reg r/m	[disp][disp]	MOV mem/reg,reg（バイト）
89	mod reg r/m	[disp][disp]	MOV mem/reg,reg（ワード）

オブジェクト・コード			ニーモニック
バイト＃0	バイト＃1	後続バイト	
8A	mod reg r/m	[disp][disp]	MOV reg,mem/reg（バイト）
8B	mod reg r/m	[disp][disp]	MOV reg,mem/reg（ワード）
8C	mod 0ss r/m	[disp][disp]	MOV mem/reg,segreg
8C	xx 1xxxxx		未使用
8D	mod reg r/m	[disp][disp]	LEA reg,addr
8E	mod 0ss r/m	[disp][disp]	MOV segreg, mem/reg
8E	xx 1xxxxx		未使用
8F	mod 000 r/m	[disp][disp]	POP mem/reg
8F	xx 001 xxx		未使用
8F	xx 010 xxx		未使用
8F	xx 011 xxx		未使用
8F	xx 100 xxx		未使用
8F	xx 101 xxx		未使用
8F	xx 110 xxx		未使用
8F	xx 111 xxx		未使用
			未使用
90			NOP
91			XCHG AX,CX
92			XCHG AX,DX
93			XCHG AX,BX
94			XCHG AX,SP
95			XCHG AX,BP
96			XCHG AX,SI
97			XCHG AX,DI
98			CBW
99			CWD
9A	kk	jj hh gg	CALL addr
9B			WAIT
9C			PUSHF
9D			POPF
9E			SAHF
9F			LAHF
A0	qq	pp	MOV AL,addr
A1	qq	pp	MOV AX,addr
A2	qq	pp	MOV addr,AL
A3	qq	pp	MOV addr,AX
A4			MOVS BYTE
A5			MOVS WORD
A6			CMPS BYTE
A7			CMPS WORD
A8	kk		TEST, AL,kk
A9	kk	jj	TEST AX,jjkk
AA			STOS BYTE
AB			STOS WORD
AC			LODS BYTE
AD			LODS WORD
AE			SCAS BYTE
AF			SCAS WORD

オブジェクト・コード			ニーモニック
バイト＃0	バイト＃1	後続バイト	
B0	kk		MOV AL,kk
B1	kk		MOV CL,kk
B2	kk		MOV DL,kk
B3	kk		MOV BL,kk
B4	kk		MOV AH,kk
B5	kk		MOV CH,kk
B6	kk		MOV DH,kk
B7	kk		MOV BH,kk
B8	kk	jj	MOV AX,jjkk
B9	kk	jj	MOV CX,jjkk
BA	kk	jj	MOV DX,jjkk
BB	kk	jj	MOV BX,jjkk
BC	kk	jj	MOV SP,jjkk
BD	kk	jj	MOV BP,jjkk
BE	kk	jj	MOV SI,jjkk
BF	kk	jj	MOV DI,jjkk
C0			未使用
C1			未使用
C2	kk	jj	RET jjkk
C3			RET
C4	mod reg r/m	[disp][disp]	LES reg,addr
C5	mod reg r/m	[disp][disp]	LDS reg,addr
C6	mod 000 r/m	[disp][disp] kk	MOV mem,kk
C6	xx 001 xxx		未使用
C6	xx 010 xxx		未使用
C6	xx 011 xxx		未使用
C6	xx 100 xxx		未使用
C6	xx 101 xxx		未使用
C6	xx 110 xxx		未使用
C6	xx 111 xxx		未使用
C7	mod 000 r/m	[disp][disp] kkjj	MOV mem,jjkk
C7	xx 001 xxx		未使用
C7	xx 010 xxx		未使用
C7	xx 011 xxx		未使用
C7	xx 100 xxx		未使用
C7	xx 101 xxx		未使用
C7	xx 110 xxx		未使用
C7	xx 111 xxx		未使用
C8			未使用
C9			未使用
CA	kk	jj	RET jjkk
CB			RET
CC			INT 3
CD	type		INT Type
CE			INTO
CF			IRET

オブジェクト・コード			ニーモニック
バイト#0	バイト#1	後続バイト	
D0	mod 000 r/m	[disp][disp]	ROL mem/reg,1(バイト)
D0	mod 001 r/m	[disp][disp]	ROR mem/reg,1(バイト)
D0	mod 010 r/m	[disp][disp]	RCL mem/reg,1(バイト)
D0	mod 011 r/m	[disp][disp]	RCR mem/reg,1(バイト)
D0	mod 100 r/m	[disp][disp]	SAL or SHL mem/reg,1(バイト)
D0	mod 101 r/m	[disp][disp]	SHR mem/reg,1(バイト)
D0	xx 110 xxx		未使用
D0	mod 111 r/m	[disp][disp]	SAR mem/reg,1(バイト)
D1	mod 000 r/m	[disp][disp]	ROL mem/reg,1(ワード)
D1	mod 001 r/m	[disp][disp]	ROR mem/reg,1(ワード)
D1	mod 010 r/m	[disp][disp]	RCL mem/reg,1(ワード)
D1	mod 011 r/m	[disp][disp]	RCR mem/reg,1(ワード)
D1	mod 100 r/m	[disp][disp]	SAL or SHL mem/reg,1(ワード)
D1	mod 101 r/m	[disp][disp]	SHR mem/reg,1(ワード)
D1	xx 110 xxx		未使用
D1	mod 111 r/m	[disp][disp]	SAR mem/reg,1(ワード)
D2	mod 000 r/m	[disp][disp]	ROL mem/reg,CL(バイト)
D2	mod 001 r/m	[disp][disp]	ROR mem/reg,CL(バイト)
D2	mod 010 r/m	[disp][disp]	RCL mem/reg,CL(バイト)
D2	mod 011 r/m	[disp][disp]	RCR mem/reg,CL(バイト)
D2	mod 100 r/m	[disp][disp]	SAL or SHL mem/reg,CL(バイト)
D2	mod 101 r/m	[disp][disp]	SHR mem/reg,CL(バイト)
D2	xx 110 xxx		未使用
D2	mod 111 r/m	[disp][disp]	SAR mem/reg,CL(バイト)
D3	mod 000 r/m	[disp][disp]	ROL mem/reg,CL(ワード)
D3	mod 001 r/m	[disp][disp]	ROR mem/reg,CL(ワード)
D3	mod 010 r/m	[disp][disp]	RCL mem/reg,CL(ワード)
D3	mod 011 r/m	[disp][disp]	RCR mem/reg,CL(ワード)
D3	mod 100 r/m	[disp][disp]	SAL or SHL mem/reg,CL(ワード)
D3	mod 101 r/m	[disp][disp]	SHR mem/reg,CL(ワード)
D3	xx 110 xxx		未使用
D3	mod 111 r/m	[disp][disp]	SAR mem/reg,CL(ワード)
D4	0A		AAM
D5	0A		AAD
D6			未使用
D7			XLAT
D8	mod xxx r/m	[disp][disp]	ESC mem/reg
D9	mod xxx r/m	[disp][disp]	ESC mem/reg
DA	mod xxx r/m	[disp][disp]	ESC mem/reg
DB	mod xxx r/m	[disp][disp]	ESC mem/reg
DC	mod xxx r/m	[disp][disp]	ESC mem/reg
DD	mod xxx r/m	[disp][disp]	ESC mem/reg
DE	mod xxx r/m	[disp][disp]	ESC mem/reg
DF	mod xxx r/m	[disp][disp]	ESC mem/reg
E0	disp		LOOPNE/LOOPNZ disp
E1	disp		LOOPE/LOOPZ disp
E2	disp		LOOP disp
E3	disp		JCXZ disp
E4	kk		IN AL,kk
E5	kk		IN AX,kk
E6	kk		OUT kk,AL
E7	kk		OUT kk,AX
E8	disp	disp	CALL disp16
E9	disp	disp	JMP disp16

オブジェクト・コード			ニーモニック
バイト＃0	バイト＃1	後続バイト	
EA	kk	jj hh gg	JMP addr
EB	disp		JMP disp
EC			IN AL,DX
ED			IN AX,DX
EE			OUT DX,AL
EF			OUT DX,AX
F0			LOCK
F1			未使用
F2			REPNE or REPNZ
F3			REP or REPE or REPZ
F4			HLT
F5			CMC
F6	mod 000 r/m	[disp][disp] kk	TEST mem/reg,kk
F6	xx 001 xxx		未使用
F6	mod 010 r/m	[disp][disp]	NOT mem/reg（バイト）
F6	mod 011 r/m	[disp][disp]	NEG mem/reg（バイト）
F6	mod 100 r/m	[disp][disp]	MUL mem/reg（バイト）
F6	mod 101 r/m	[disp][disp]	IMUL mem/reg（バイト）
F6	mod 110 r/m	[disp][disp]	DIV mem/reg（バイト）
F6	mod 111 r/m	[disp][disp]	IDIV mem/reg（バイト）
F7	mod 000 r/m	[disp][disp] kkjj	TEST mem/reg,jjkk
F7	xx 001 xxx		未使用
F7	mod 010 r/m	[disp][disp]	NOT mem/reg（ワード）
F7	mod 011 r/m	[disp][disp]	NEG mem/reg（ワード）
F7	mod 100 r/m	[disp][disp]	MUL mem/reg（ワード）
F7	mod 101 r/m	[disp][disp]	IMUL mem/reg（ワード）
F7	mod 110 r/m	[disp][disp]	DIV mem/reg（ワード）
F7	mod 111 r/m	[disp][disp]	IDIV mem/reg（ワード）
F8			CLC
F9			STC
FA			CLI
FB			STI
FC			CLD
FD			STD
FE	mod 000 r/m	[disp][disp]	INC mem/reg（バイト）
FE	mod 001 r/m	[disp][disp]	DEC mem/reg（バイト）
FE	xx 010 xxx		未使用
FE	xx 011 xxx		未使用
FE	xx 100 xxx		未使用
FE	xx 101 xxx		未使用
FE	xx 110 xxx		未使用
FE	xx 111 xxx		未使用
FF	mod 000 r/m	[disp][disp]	INC mem/reg（ワード）
FF	mod 001 r/m	[disp][disp]	DEC mem/reg（ワード）
FF	mod 010 r/m	[disp][disp]	CALL mem/reg
FF	mod 011 r/m	[disp][disp]	CALL mem
FF	mod 100 r/m	[disp][disp]	JMP mem/reg
FF	mod 101 r/m	[disp][disp]	JMP mem
FF	mod 110 r/m	[disp][disp]	PUSH mem
FF	xx 111 xxx		未使用

付録C

8086と8088ファミリーの AC,DC特性と信号波形

ここでは，原著のものではなく，Intelの“Component Data Catalog”, January 1982から許可を得て，訳者が判断して選択したものが掲載されている．

掲載内容は以下に示す素子についての電気的特性とタイミング・データであり，ほぼ原著のものと同じである．

- 8086CPU
- 8088CPU
- 8282／8283オクタル・ラッチ
- 8284Aクロック・ジェネレータ
- 8286／8287オクタル・バス・トランシーバ
- 8288バス・コントローラ
- 8289バス・アービタ
- その他

iAPX 86/10
16-BIT HMOS MICROPROCESSOR

8086/8086-2/8086-1

- **Direct Addressing Capability to 1 MByte of Memory**
- **Architecture Designed for Powerful Assembly Language and Efficient High Level Languages.**
- **14 Word, by 16-Bit Register Set with Symmetrical Operations**
- **24 Operand Addressing Modes**
- **Bit, Byte, Word, and Block Operations**
- **8 and 16-Bit Signed and Unsigned Arithmetic in Binary or Decimal Including Multiply and Divide**
- **Range of Clock Rates:**
  5 MHz for 8086,
  8 MHz for 8086-2,
  10 MHz for 8086-1
- **MULTIBUS™ System Compatible Interface**

The Intel iAPX 86/10 high performance 16-bit CPU is available in three clock rates: 5, 8 and 10 MHz. The CPU is implemented in N-Channel, depletion load, silicon gate technology (HMOS), and packaged in a 40-pin CerDIP package. The iAPX 86/10 operates in both single processor and multiple processor configurations to achieve high performance levels.

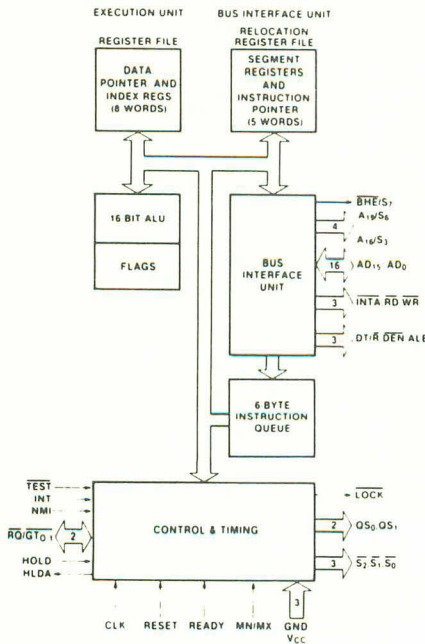



Figure 1. iAPX 86/10 CPU Block Diagram

Pin	Name	Pin	Name
1	GND	40	Vcc
2	AD14	39	AD15
3	AD13	38	A16/S3
4	AD12	37	A17/S4
5	AD11	36	A18/S5
6	AD10	35	A19/S6
7	AD9	34	BHE/S7
8	AD8	33	MN/MX
9	AD7	32	RD
10	AD6	31	RQ/GT0 (HOLD)
11	AD5	30	RQ/GT1 (HLDA)
12	AD4	29	LOCK (WR)
13	AD3	28	S2 (M/IO)
14	AD2	27	S1 (DT/R)
15	AD1	26	S0 (DEN)
16	AD0	25	QS0 (ALE)
17	NMI	24	QS1 (INTA)
18	INTR	23	TEST
19	CLK	22	READY
20	GND	21	RESET

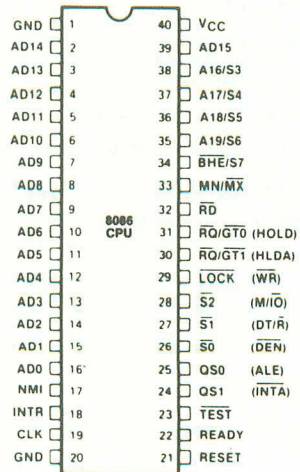
8086 CPU

40 LEAD

Figure 2. iAPX 86/10 Pin Configuration

ABSOLUTE MAXIMUM RATINGS*

Ambient Temperature Under Bias ....0°C to 70°C
Storage Temperature .... −65°C to +150°C
Voltage on Any Pin with
  Respect to Ground .... −1.0 to +7V
Power Dissipation .... 2.5 Watt

**NOTICE: Stresses above those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.*

D.C. CHARACTERISTICS

(8086: T_A = 0°C to 70°C, V_{CC} = 5V ± 10%)
(8086-1: T_A = 0°C to 70°C, V_{CC} = 5V ± 5%)
(8086-2: T_A = 0°C to 70°C, V_{CC} = 5V ± 5%)

Symbol	Parameter	Min.	Max.	Units	Test Conditions
V_{IL}	Input Low Voltage	−0.5	+0.8	V	
V_{IH}	Input High Voltage	2.0	V_{CC}+0.5	V	
V_{OL}	Output Low Voltage		0.45	V	I_{OL}=2.5 mA
V_{OH}	Output High Voltage	2.4		V	I_{OH}= −400 μA
I_{CC}	Power Supply Current: 8086 8086-1 8086-2		340 360 350	mA	T_A=25°C
I_{LI}	Input Leakage Current		±10	μA	0V ≤ V_{IN} ≤ V_{CC}
I_{LO}	Output Leakage Current		±10	μA	0.45V ≤ V_{OUT} ≤ V_{CC}
V_{CL}	Clock Input Low Voltage	−0.5	+0.6	V	
V_{CH}	Clock Input High Voltage	3.9	V_{CC}+1.0	V	
C_{IN}	Capacitance of Input Buffer (All input except AD_0 − AD_{15}, $\overline{RQ}/\overline{GT}$)		15	pF	fc=1 MHz
C_{IO}	Capacitance of I/O Buffer (AD_0 − AD_{15}, $\overline{RQ}/\overline{GT}$)		15	pF	fc=1 MHz

A.C. CHARACTERISTICS

(8086: T_A = 0°C to 70°C, V_{CC} = 5V ± 10%)
(8086-1: T_A = 0°C to 70°C, V_{CC} = 5V ± 5%)
(8086-2: T_A = 0°C to 70°C, V_{CC} = 5V ± 5%)

MINIMUM COMPLEXITY SYSTEM TIMING REQUIREMENTS

Symbol	Parameter	8086		8086-1 (Preliminary)		8086-2		Units	Test Conditions
		Min.	Max.	Min.	Max.	Min.	Max.		
TCLCL	CLK Cycle Period	200	500	100	500	125	500	ns	
TCLCH	CLK Low Time	118		53		68		ns	
TCHCL	CLK High Time	69		39		44		ns	
TCH1CH2	CLK Rise Time		10		10		10	ns	From 1.0V to 3.5V
TCL2CL1	CLK Fall Time		10		10		10	ns	From 3.5V to 1.0V
TDVCL	Data in Setup Time	30		5		20		ns	
TCLDX	Data in Hold Time	10		10		10		ns	
TR1VCL	RDY Setup Time into 8284A (See Notes 1, 2)	35		35		35		ns	
TCLR1X	RDY Hold Time into 8284A (See Notes 1, 2)	0		0		0		ns	
TRYHCH	READY Setup Time into 8086	118		53		68		ns	
TCHRYX	READY Hold Time into 8086	30		20		20		ns	
TRYLCL	READY Inactive to CLK (See Note 3)	−8		−10		−8		ns	
THVCH	HOLD Setup Time	35		20		20		ns	
TINVCH	INTR, NMI, $\overline{\text{TEST}}$ Setup Time (See Note 2)	30		15		15		ns	
TILIH	Input Rise Time (Except CLK)		20		20		20	ns	From 0.8V to 2.0V
TIHIL	Input Fall Time (Except CLK)		12		12		12	ns	From 2.0V to 0.8V

A.C. CHARACTERISTICS (Continued)

TIMING RESPONSES

Symbol	Parameter	8086		8086-1 (Preliminary)		8086-2		Units	Test Conditions
		Min.	Max.	Min.	Max.	Min.	Max.		
TCLAV	Address Valid Delay	10	110	10	50	10	60	ns	
TCLAX	Address Hold Time	10		10		10		ns	
TCLAZ	Address Float Delay	TCLAX	80	10	40	TCLAX	50	ns	
TLHLL	ALE Width	TCLCH−20		TCLCH−10		TCLCH-10		ns	
TCLLH	ALE Active Delay		80		40		50	ns	
TCHLL	ALE Inactive Delay		85		45		55	ns	
TLLAX	Address Hold Time to ALE Inactive	TCHCL−10		TCHCL−10		TCHCL−10		ns	
TCLDV	Data Valid Delay	10	110	10	50	10	60	ns	*C_L = 20-100 pF for all 8086 Outputs (In addition to 8086 self-load)
TCHDX	Data Hold Time	10		10		10		ns	
TWHDX	Data Hold Time After WR	TCLCH−30		TCLCH−25		TCLCH−30		ns	
TCVCTV	Control Active Delay 1	10	110	10	50	10	70	ns	
TCHCTV	Control Active Delay 2	10	110	10	45	10	60	ns	
TCVCTX	Control Inactive Delay	10	110	10	50	10	70	ns	
TAZRL	Address Float to READ Active	0		0		0		ns	
TCLRL	$\overline{RD}$ Active Delay	10	165	10	70	10	100	ns	
TCLRH	$\overline{RD}$ Inactive Delay	10	150	10	60	10	80	ns	
TRHAV	$\overline{RD}$ Inactive to Next Address Active	TCLCL−45		TCLCL−35		TCLCL−40		ns	
TCLHAV	HLDA Valid Delay	10	160	10	60	10	100	ns	
TRLRH	$\overline{RD}$ Width	2TCLCL−75		2TCLCL−40		2TCLCL−50		ns	
TWLWH	$\overline{WR}$ Width	2TCLCL−60		2TCLCL−35		2TCLCL−40		ns	
TAVAL	Address Valid to ALE Low	TCLCH−60		TCLCH−35		TCLCH−40		ns	
TOLOH	Output Rise Time		20		20		20	ns	From 0.8V to 2.0V
TOHOL	Output Fall Time		12		12		12	ns	From 2.0V to 0.8V

NOTES:
1. Signal at 8284A shown for reference only.
2. Setup requirement for asynchronous signal only to guarantee recognition at next CLK.
3. Applies only to T2 state. (8 ns into T3).

A.C. TESTING INPUT, OUTPUT WAVEFORM

A.C. TESTING: INPUTS ARE DRIVEN AT 2.4V FOR A LOGIC "1" AND 0.45V FOR A LOGIC "0". TIMING MEASUREMENTS ARE MADE AT 1.5V FOR BOTH A LOGIC "1" AND "0".

A.C. TESTING LOAD CIRCUIT

C_L INCLUDES JIG CAPACITANCE

WAVEFORMS

MINIMUM MODE

CLK (8284A Output)

M/$\overline{IO}$

$\overline{BHE}$/S7, A_{19}/S_6–A_{16}/S_3

ALE

RDY (8284A Input) SEE NOTE 4

READY (8086 Input)

READ CYCLE (NOTE 1) ($\overline{WR}$, $\overline{INTA}$ = V_{OH})

AD_{15}–AD_0

$\overline{RD}$

DT/$\overline{R}$

$\overline{DEN}$

WAVEFORMS (Continued)

NOTES:

1. All signals switch between V_{OH} and V_{OL} unless otherwise specified.
2. RDY is sampled near the end of T_2, T_3, T_W to determine if T_W machines states are to be inserted.
3. Two INTA cycles run back-to-back. The 8086 LOCAL ADDR/DATA BUS is floating during both INTA cycles. Control signals shown for second INTA cycle.
4. Signals at 8284A are shown for reference only.
5. All timing measurements are made at 1.5V unless otherwise noted.

A.C. CHARACTERISTICS

MAX MODE SYSTEM (USING 8288 BUS CONTROLLER)
TIMING REQUIREMENTS

Symbol	Parameter	8086		8086-1 (Preliminary)		8086-2 (Preliminary)		Units	Test Conditions
		Min.	Max.	Min.	Max.	Min.	Max.		
TCLCL	CLK Cycle Period	200	500	100	500	125	500	ns	
TCLCH	CLK Low Time	118		53		68		ns	
TCHCL	CLK High Time	69		39		44		ns	
TCH1CH2	CLK Rise Time		10		10		10	ns	From 1.0V to 3.5V
TCL2CL1	CLK Fall Time		10		10		10	ns	From 3.5V to 1.0V
TDVCL	Data in Setup Time	30		5		20		ns	
TCLDX	Data In Hold Time	10		10		10		ns	
TR1VCL	RDY Setup Time into 8284A (See Notes 1, 2)	35		35		35		ns	
TCLR1X	RDY Hold Time into 8284A (See Notes 1, 2)	0		0		0		ns	
TRYHCH	READY Setup Time into 8086	118		53		68		ns	
TCHRYX	READY Hold Time into 8086	30		20		20		ns	
TRYLCL	READY Inactive to CLK (See Note 4)	−8		−10		−8		ns	
TINVCH	Setup Time for Recognition (INTR, NMI, $\overline{\text{TEST}}$) (See Note 2)	30		15		15		ns	
TGVCH	$\overline{\text{RQ}}/\overline{\text{GT}}$ Setup Time	30		12		15		ns	
TCHGX	$\overline{\text{RQ}}$ Hold Time into 8086	40		20		30		ns	
TILIH	Input Rise Time (Except CLK)		20		20		20	ns	From 0.8V to 2.0V
TIHIL	Input Fall Time (Except CLK)		12		12		12	ns	From 2.0V to 0.8V

NOTES:
1. Signal at 8284A or 8288 shown for reference only.
2. Setup requirement for asynchronous signal only to guarantee recognition at next CLK.
3. Applies only to T3 and wait states.
4. Applies only to T2 state (8 ns into T3).

A.C. CHARACTERISTICS (Continued)

TIMING RESPONSES

Symbol	Parameter	8086		8086-1 (Preliminary)		8086-2 (Preliminary)		Units	Test Conditions
		Min.	Max.	Min.	Max.	Min.	Max.		
TCLML	Command Active Delay (See Note 1)	10	35	10	35	10	35	ns	
TCLMH	Command Inactive Delay (See Note 1)	10	35	10	35	10	35	ns	
TRYHSH	READY Active to Status Passive (See Note 3)		110		45		65	ns	
TCHSV	Status Active Delay	10	110	10	45	10	60	ns	
TCLSH	Status Inactive Delay	10	130	10	55	10	70	ns	
TCLAV	Address Valid Delay	10	110	10	50	10	60	ns	
TCLAX	Address Hold Time	10		10		10		ns	
TCLAZ	Address Float Delay	TCLAX	80	10	40	TCLAX	50	ns	
TSVLH	Status Valid to ALE High (See Note 1)		15		15		15	ns	
TSVMCH	Status Valid to MCE High (See Note 1)		15		15		15	ns	
TCLLH	CLK Low to ALE Valid (See Note 1)		15		15		15	ns	
TCLMCH	CLK Low to MCE High (See Note 1)		15		15		15	ns	
TCHLL	ALE Inactive Delay (See Note 1)		15		15		15	ns	C_L = 20-100 pF for all 8086 Outputs (In addition to 8086 self-load)
TCLMCL	MCE Inactive Delay (See Note 1)		15		15		15	ns	
TCLDV	Data Valid Delay	10	110	10	50	10	60	ns	
TCHDX	Data Hold Time	10		10		10		ns	
TCVNV	Control Active Delay (See Note 1)	5	45	5	45	5	45	ns	
TCVNX	Control Inactive Delay (See Note 1)	10	45	10	45	10	45	ns	
TAZRL	Address Float to Read Active	0		0		0		ns	
TCLRL	RD Active Delay	10	165	10	70	10	100	ns	
TCLRH	RD Inactive Delay	10	150	10	60	10	80	ns	
TRHAV	RD Inactive to Next Address Active	TCLCL−45		TCLCL−35		TCLCL−40		ns	
TCHDTL	Direction Control Active Delay (See Note 1)		50		50		50	ns	
TCHDTH	Direction Control Inactive Delay (See Note 1)		30		30		30	ns	
TCLGL	GT Active Delay	0	85	0	45	0	50	ns	
TCLGH	GT Inactive Delay	0	85	0	45	0	50	ns	
TRLRH	RD Width	2TCLCL−75		2TCLCL−40		2TCLCL−50		ns	
TOLOH	Output Rise Time		20		20		20	ns	From 0.8V to 2.0V
TOHOL	Output Fall Time		12		12		12	ns	From 2.0V to 0.8V

WAVEFORMS

MAXIMUM MODE
T1
T2
T3
T4
Tw
TCLCL
TCH1CH2
TCL2CL1
CLK
VCH
VCL
TCLAV
TCHCL
TCLCH
QS0,QS1
TCHSV
TCLSH
S2,S1,S0 (EXCEPT HALT)
(SEE NOTE 8)
TCLAV
TCLAX
TCLDV
TCHDX
BHE/S7, A19/S6-A16/S3
BHE, A19-A16
S7-S3
TSVLH
TCLLH
TCHLL
SEE NOTE 5
ALE (8288 OUTPUT)
RDY (8284A INPUT)
TR1VCL
TCLR1X
TRYLCL
READY (8086 INPUT)
TCHRYX
TRYHSH
TCLAX
TRYHCH
READ CYCLE
TCLAV
TCLAZ
TDVCL
TCLDX
AD15-AD0
A15-AD0
FLOAT
DATA IN
FLOAT
TAZRL
TCLRH
TRHAV
RD
TCHDTL
TCLRL
TRLRH
TCHDTH
DT/R
TCLML
TCLMH
8288 OUTPUTS
SEE NOTES 5,6
MRDC OR IORC
TCVNV
DEN
TCVNX

WAVEFORMS (Continued)

NOTES:

1. All signals switch between V_{OH} and V_{OL} unless otherwise specified.
2. RDY is sampled near the end of T_2, T_3, T_W to determine if T_W machines states are to be inserted.
3. Cascade address is valid between first and second INTA cycle.
4. Two INTA cycles run back-to-back. The 8086 LOCAL ADDR/DATA BUS is floating during both INTA cycles. Control for pointer address is shown for second INTA cycle.
5. Signals at 8284A or 8288 are shown for reference only.
6. The issuance of the 8288 command and control signals ($\overline{MRDC}$, $\overline{MWTC}$, $\overline{AMWC}$, $\overline{IORC}$, $\overline{IOWC}$, $\overline{AIOWC}$, $\overline{INTA}$ and DEN) lags the active high 8288 CEN.
7. All timing measurements are made at 1.5V unless otherwise noted.
8. Status inactive in state just prior to T_4.

WAVEFORMS (Continued)

iAPX 88/10
8-BIT HMOS MICROPROCESSOR
8088/8088-2

- 8-Bit Data Bus Interface
- 16-Bit Internal Architecture
- Direct Addressing Capability to 1 Mbyte of Memory
- Direct Software Compatibility with iAPX 86/10 (8086 CPU)
- 14-Word by 16-Bit Register Set with Symmetrical Operations
- 24 Operand Addressing Modes
- Byte, Word, and Block Operations
- 8-Bit and 16-Bit Signed and Unsigned Arithmetic in Binary or Decimal, Including Multiply and Divide
- Compatible with 8155-2, 8755A-2 and 8185-2 Multiplexed Peripherals
- Two Clock Rates:
  5 MHz for 8088
  8 MHz for 8088-2

The Intel® iAPX 88/10 is a new generation, high performance microprocessor implemented in N-channel, depletion load, silicon gate technology (HMOS), and packaged in a 40-pin CerDIP package. The processor has attributes of both 8- and 16-bit microprocessors. It is directly compatible with iAPX 86/10 software and 8080/8085 hardware and peripherals.

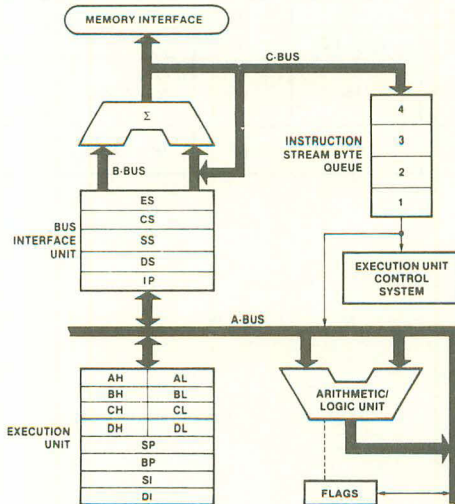



Figure 1. iAPX 88/10 CPU Functional Block Diagram

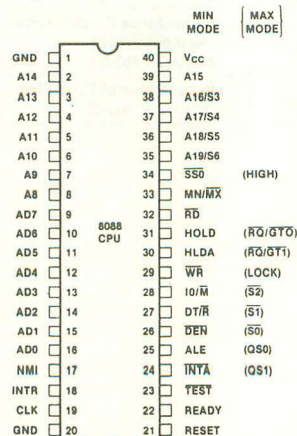



Figure 2. iAPX 88/10 Pin Configuration

ABSOLUTE MAXIMUM RATINGS*

Ambient Temperature Under Bias ....0°C to 70°C
Storage Temperature............. −65°C to +150°C
Voltage on Any Pin with
Respect to Ground.................... −1.0 to +7V
Power Dissipation .... 2.5 Watt

**NOTICE: Stresses above those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.*

D.C. CHARACTERISTICS

(8088: T_A = 0°C to 70°C, V_{CC} = 5V ±10%)
(8088-2: T_A = 0°C to 70°C, V_{CC} = 5V ±5%)

Symbol	Parameter	Min.	Max.	Units	Test Conditions
V_{IL}	Input Low Voltage	−0.5	+0.8	V	
V_{IH}	Input High Voltage	2.0	V_{CC}+0.5	V	
V_{OL}	Output Low Voltage		0.45	V	I_{OL} = 2.0 mA
V_{OH}	Output High Voltage	2.4		V	I_{OH} = −400 μA
I_{CC}	Power Supply Current: 8088 8088-2		340 350	mA	T_A = 25°C
I_{LI}	Input Leakage Current		±10	μA	0V ≤ V_{IN} ≤ V_{CC}
I_{LO}	Output Leakage Current		±10	μA	0.45V ≤ V_{OUT} ≤ V_{CC}
V_{CL}	Clock Input Low Voltage	−0.5	+0.6	V	
V_{CH}	Clock Input High Voltage	3.9	V_{CC}+1.0	V	
C_{IN}	Capacitance if Input Buffer (All input except AD_0-AD_7, RQ/GT		15	pF	fc = 1 MHz
C_{IO}	Capacitance of I/O Buffer (AD_0-AD_7, RQ/GT		15	pF	fc = 1 MHz

A.C. CHARACTERISTICS (8088: T_A = 0°C to 70°C, V_{CC} = 5V ±10%) (8088-2: T_A = 0°C to 70°C, V_{CC} = 5V ±5%)

MINIMUM COMPLEXITY SYSTEM TIMING REQUIREMENTS

Symbol	Parameter	8088 Min.	8088 Max.	8088-2 Min.	8088-2 Max.	Units	Test Conditions
TCLCL	CLK Cycle Period	200	500	125	500	ns	
TCLCH	CLK Low Time	(⅔TCLCL)−15		(⅔TCLCL)−15		ns	
TCHCL	CLK High Time	(⅓TCLCL)+2		(⅓TCLCL)+2		ns	
TCH1CH2	CLK Rise Time		10		10	ns	From 1.0V to 3.5V
TCL2CL1	CLK Fall Time		10		10	ns	From 3.5V to 1.0V
TDVCL	Data in Setup Time	30		20		ns	
TCLDX	Data in Hold Time	10		10		ns	
TR1VCL	RDY Setup Time into 8284 (See Notes 1, 2)	35		35		ns	
TCLR1X	RDY Hold Time into 8284 (See Notes 1, 2)	0		0		ns	
TRYHCH	READY Setup Time into 8088	(⅔TCLCL)−15		(⅔TCLCL)−15		ns	
TCHRYX	READY Hold Time into 8088	30		20		ns	
TRYLCL	READY Inactive to CLK (See Note 3)	−8		−8		ns	
THVCH	HOLD Setup Time	35		20		ns	
TINVCH	INTR, NMI, $\overline{TEST}$ Setup Time (See Note 2)	30		15		ns	
TILIH	Input Rise Time (Except CLK)		20		20	ns	From 0.8V to 2.0V
TIHIL	Input Fall Time (Except CLK)		12		12	ns	From 2.0V to 0.8V

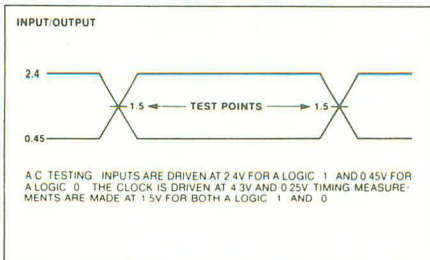
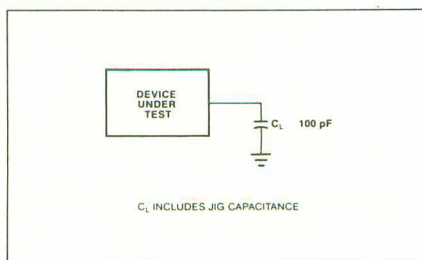
A.C. CHARACTERISTICS (Continued)

TIMING RESPONSES

Symbol	Parameter	8088 Min.	8088 Max.	8088-2 Min.	8088-2 Max.	Units	Test Conditions
TCLAV	Address Valid Delay	10	110	10	60	ns	C_L = 20-100 pF for all 8088 Outputs in addition to internal loads
TCLAX	Address Hold Time	10		10		ns	
TCLAZ	Address Float Delay	TCLAX	80	TCLAX	50	ns	
TLHLL	ALE Width	TCLCH−20		TCLCH−10		ns	
TCLLH	ALE Active Delay		80		50	ns	
TCHLL	ALE Inactive Delay		85		55	ns	
TLLAX	Address Hold Time to ALE Inactive	TCHCL−10		TCHCL−10		ns	
TCLDV	Data Valid Delay	10	110	10	60	ns	
TCHDX	Data Hold Time	10		10		ns	
TWHDX	Data Hold Time After $\overline{WR}$	TCLCH−30		TCLCH−30		ns	
TCVCTV	Control Active Delay 1	10	110	10	70	ns	
TCHCTV	Control Active Delay 2	10	110	10	60	ns	
TCVCTX	Control Inactive Delay	10	110	10	70	ns	
TAZRL	Address Float to READ Active	0		0		ns	
TCLRL	$\overline{RD}$ Active Delay	10	165	10	100	ns	
TCLRH	$\overline{RD}$ Inactive Delay	10	150	10	80	ns	
TRHAV	$\overline{RD}$ Inactive to Next Address Active	TCLCL−45		TCLCL−40		ns	
TCLHAV	HLDA Valid Delay	10	160	10	100	ns	
TRLRH	$\overline{RD}$ Width	2TCLCL−75		2TCLCL−50		ns	
TWLWH	$\overline{WR}$ Width	2TCLCL−60		2TCLCL−40		ns	
TAVAL	Address Valid to ALE Low	TCLCH−60		TCLCH−40		ns	
TOLOH	Output Rise Time		20		20	ns	From 0.8V to 2.0V
TOHOL	Output Fall Time		12		12	ns	From 2.0V to 0.8V

A.C. TESTING INPUT, OUTPUT WAVEFORM

A.C. TESTING LOAD CIRCUIT

WAVEFORMS

BUS TIMING—MINIMUM MODE SYSTEM
T1
T2
T3
Tw
T4
TCLCL
TCH1CH2
TCL2CL1
VCH
CLK (8284 Output)
VCL
TCHCTV
TCHCL
TCLCH
IO/M, SS0
A15–A8
A15 – A8 (Float during INTA)
TCLAV
TCLAX
TCLDV
TCHDX
A19/S6–A16/S3
A19–A16
S7–S3
TCLLH
TLHLL
TLLAX
ALE
TCHLL
TR1VCL
TAVAL
VIH
RDY (8284 Input)
SEE NOTE 5
VIL
TCLR1X
TRYLCL
READY (8088 Input)
TCHRYX
TRYHCH
TCLAZ
TDVCL
TCLDX
AD7 – AD0
AD7–AD0
FLOAT
DATA IN
FLOAT
TAZRL
TCLRH
TRHAV
RD
READ CYCLE
(NOTE 1)
(WR, INTA = VOH)
TCHCTV
TCLRL
TRLRH
TCHCTV
DT/R
TCVCTV
TCVCTX
DEN

WAVEFORMS (Continued)

A.C. CHARACTERISTICS

MAX MODE SYSTEM (USING 8288 BUS CONTROLLER)

TIMING REQUIREMENTS

Symbol	Parameter	8088		8088-2		Units	Test Conditions
		Min.	Max.	Min.	Max.		
TCLCL	CLK Cycle Period	200	500	125	500	ns	
TCLCH	CLK Low Time	(⅔ TCLCL) − 15		(⅔ TCLCL) − 15		ns	
TCHCL	CLK High Time	(⅓ TCLCL) + 2		(⅓ TCLCL) + 2		ns	
TCH1CH2	CLK Rise Time		10		10	ns	From 1.0V to 3.5V
TCL2CL1	CLK Fall Time		10		10	ns	From 3.5V to 1.0V
TDVCL	Data In Setup Time	30		20		ns	
TCLDX	Data In Hold Time	10		10		ns	
TR1VCL	RDY Setup Time into 8284 (See Notes 1, 2)	35		35		ns	
TCLR1X	RDY Hold Time into 8284 (See Notes 1, 2)	0		0		ns	
TRYHCH	READY Setup Time into 8088	(⅔ TCLCL) − 15		(⅔ TCLCL) − 15		ns	
TCHRYX	READY Hold Time into 8088	30		20		ns	
TRYLCL	READY Inactive to CLK (See Note 4)	−8		−8		ns	
TINVCH	Setup Time for Recognition (INTR, NMI, TEST) (See Note 2)	30		15		ns	
TGVCH	RQ/GT Setup Time	30		15		ns	
TCHGX	RQ Hold Time into 8086	40		30		ns	
TILIH	Input Rise Time (Except CLK)		20		20	ns	From 0.8V to 2.0V
TIHIL	Input Fall Time (Except CLK)		12		12	ns	From 2.0V to 0.8V

NOTES:
1. Signal at 8284 or 8288 shown for reference only.
2. Setup requirement for asynchronous signal only to guarantee recognition at next CLK.
3. Applies only to T2 state (8 ns into T3 state).
4. Applies only to T2 state (8 ns into T3 state).

A.C. CHARACTERISTICS

TIMING RESPONSES

Symbol	Parameter	8088 Min.	8088 Max.	8088-2 Min.	8088-2 Max.	Units	Test Conditions
TCLML	Command Active Delay (See Note 1)	10	35	10	35	ns	
TCLMH	Command Inactive Delay (See Note 1)	10	35	10	35	ns	
TRYHSH	READY Active to Status Passive (See Note 3)		110		65	ns	
TCHSV	Status Active Delay	10	110	10	60	ns	
TCLSH	Status Inactive Delay	10	130	10	70	ns	
TCLAV	Address Valid Delay	10	110	10	60	ns	
TCLAX	Address Hold Time	10		10		ns	C_L = 20-100 pF for all 8088 Outputs in addition to internal loads
TCLAZ	Address Float Delay	TCLAX	80	TCLAX	50	ns	
TSVLH	Status Valid to ALE High (See Note 1)		15		15	ns	
TSVMCH	Status Valid to MCE High (See Note 1)		15		15	ns	
TCLLH	CLK Low to ALE Valid (See Note 1)		15		15	ns	
TCLMCH	CLK Low to MCE High (See Note 1)		15		15	ns	
TCHLL	ALE Inactive Delay (See Note 1)		15		15	ns	
TCLMCL	MCE Inactive Delay (See Note 1)		15		15	ns	
TCLDV	Data Valid Delay	10	110	10	60	ns	
TCHDX	Data Hold Time	10		10		ns	
TCVNV	Control Active Delay (See Note 1)	5	45	5	45	ns	
TCVNX	Control Inactive Delay (See Note 1)	10	45	10	45	ns	
TAZRL	Address Float to Read Active	0		0		ns	
TCLRL	RD Active Delay	10	165	10	100	ns	
TCLRH	RD Inactive Delay	10	150	10	80	ns	
TRHAV	RD Inactive to Next Address Active	TCLCL−45		TCLCL−40		ns	
TCHDTL	Direction Control Active Delay (See Note 1)		50		50	ns	
TCHDTH	Direction Control Inactive Delay (See Note 1)		30		30	ns	
TCLGL	GT Active Delay		110		50	ns	
TCLGH	GT Inactive Delay		85		50	ns	
TRLRH	RD Width	2TCLCL−75		2TCLCL−50		ns	
TOLOH	Output Rise Time		20		20	ns	From 0.8V to 2.0V
TOHOL	Output Fall Time		12		12	ns	From 2.0V to 0.8V

WAVEFORMS

BUS TIMING—MAXIMUM MODE
T1
T2
T3
T4
TW
TCLCL
TCH1CH2
TCL2CL1
VCH
VCL
CLK
TCLAV
TCHCL
TCLCH
QS0,QS1
TCHSV
TCLSH
S2,S1,S0 (EXCEPT HALT)
(SEE NOTE 8)
A15 – A8
TCLAV
TCLAX
TCLDV
TCHDX
A19/S6-A16/S3
A19-A16
S7-S3
TSVLH
TCLLH
TCHLL
ALE (8288 OUTPUT)
SEE NOTE 5
RDY (8284 INPUT)
TR1VCL
TCLR1X
TRYLCL
READY (8088 INPUT)
TCHRYX
TRYHSH
TCLAX
TRYHCH
READ CYCLE
TCLAV
TCLAZ
TDVCL
TCLDX
AD7 – AD0
FLOAT
DATA IN
TAZRL
TCLRH
TRHAV
RD
TCHDTL
TCLRL
TRLRH
TCHDTH
DT/R
TCLML
TCLMH
8288 OUTPUTS
SEE NOTES 5,6
MRDC OR IORC
TCVNV
DEN
TCVNX

WAVEFORMS (Continued)

NOTES:
1. ALL SIGNALS SWITCH BETWEEN V_{OH} AND V_{OL} UNLESS OTHERWISE SPECIFIED.
2. RDY IS SAMPLED NEAR THE END OF T_2, T_3, T_W TO DETERMINE IF T_W MACHINES STATES ARE TO BE INSERTED.
3. CASCADE ADDRESS IS VALID BETWEEN FIRST AND SECOND INTA CYCLES.
4. TWO INTA CYCLES RUN BACK-TO-BACK. THE 8088 LOCAL ADDR/DATA BUS IS FLOATING DURING BOTH INTA CYCLES. CONTROL FOR POINTER ADDRESS IS SHOWN FOR SECOND INTA CYCLE.
5. SIGNALS AT 8284 OR 8288 ARE SHOWN FOR REFERENCE ONLY.
6. THE ISSUANCE OF THE 8288 COMMAND AND CONTROL SIGNALS (MRDC, MWTC, AMWC, IORC, IOWC, AIOWC, INTA AND DEN) LAGS THE ACTIVE HIGH 8288 CEN.
7. ALL TIMING MEASUREMENTS ARE MADE AT 1.5V UNLESS OTHERWISE NOTED.
8. STATUS INACTIVE IN STATE JUST PRIOR TO T_4.

WAVEFORMS (Continued)

8282/8283
OCTAL LATCH

- **Address Latch for iAPX 86, 88, MCS-80®, MCS-85®, MCS-48® Families**
- **High Output Drive Capability for Driving System Data Bus**
- **Fully Parallel 8-Bit Data Register and Buffer**
- **Transparent during Active Strobe**
- **3-State Outputs**
- **20-Pin Package with 0.3" Center**
- **No Output Low Noise when Entering or Leaving High Impedance State**

The 8282 and 8283 are 8-bit bipolar latches with 3-state output buffers. They can be used to implement latches, buffers, or multiplexers. The 8283 inverts the input data at its outputs while the 8282 does not. Thus, all of the principal peripheral and input/output functions of a microcomputer system can be implemented with these devices.

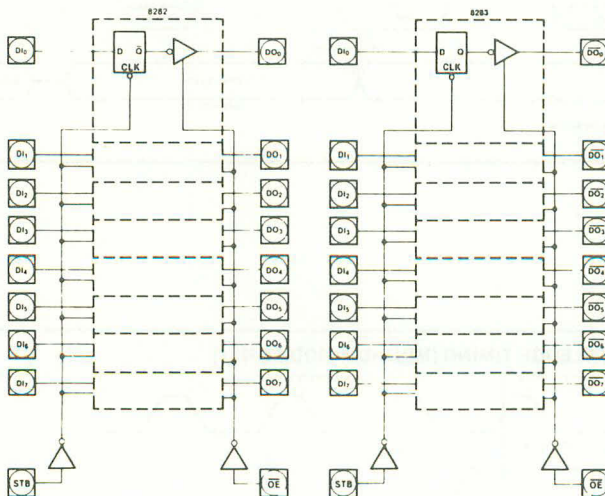



Figure 1. Logic Diagrams

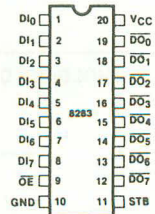
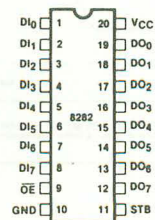



Figure 2. Pin Configurations

ABSOLUTE MAXIMUM RATINGS*

Temperature Under Bias . . . . 0°C to 70°C
Storage Temperature . . . . −65°C to +150°C
All Output and Supply Voltages . . . . −0.5V to +7V
All Input Voltages . . . . −1.0V to +5.5V
Power Dissipation . . . . 1 Watt

**NOTICE: Stresses above those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.*

D.C. CHARACTERISTICS (V_{CC} = 5V ±10%, T_A = 0°C to 70°C)

Symbol	Parameter	Min.	Max.	Units	Test Conditions
V_C	Input Clamp Voltage		−1	V	I_C = −5 mA
I_{CC}	Power Supply Current		160	mA	
I_F	Forward Input Current		−0.2	mA	V_F = 0.45V
I_R	Reverse Input Current		50	μA	V_R = 5.25V
V_{OL}	Output Low Voltage		.45	V	I_{OL} = 32 mA
V_{OH}	Output High Voltage	2.4		V	I_{OH} = −5 mA
I_{OFF}	Output Off Current		±50	μA	V_{OFF} = 0.45 to 5.25V
V_{IL}	Input Low Voltage		0.8	V	V_{CC} = 5.0V See Note 1
V_{IH}	Input High Voltage	2.0		V	V_{CC} = 5.0V See Note 1
C_{IN}	Input Capacitance		12	pF	F = 1 MHz V_{BIAS} = 2.5V, V_{CC} = 5V T_A = 25°C

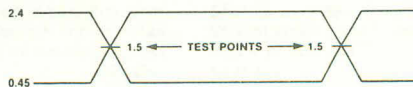
NOTE:
1. Output Loading I_{OL} = 32 mA, I_{OH} = −5 mA, C_L = 300 pF.*

A.C. CHARACTERISTICS (V_{CC} = 5V ±10%, T_A = 0°C to 70°C Loading: Outputs—I_{OL} = 32 mA, I_{OH} = −5 mA, C_L = 300 pF*)

Symbol	Parameter	Min.	Max.	Units	Test Conditions
TIVOV	Input to Output Delay —Inverting —Non-Inverting	 5 5	 22 30	 ns ns	(See Note 1)
TSHOV	STB to Output Delay —Inverting —Non-Inverting	 10 10	 40 45	 ns ns	
TEHOZ	Output Disable Time	5	18	ns	
TELOV	Output Enable Time	10	30	ns	
TIVSL	Input to STB Setup Time	0		ns	
TSLIX	Input to STB Hold Time	25		ns	
TSHSL	STB High Time	15		ns	
TILIH, TOLOH	Input, Output Rise Time		20	ns	From 0.8V to 2.0V
TIHIL, TOHOL	Input, Output Fall Time		12	ns	From 2.0V to 0.8V

NOTE:
1. See waveforms and test load circuit on following page.

*C_L = 200 pF for plastic 8282/8283.

A.C. TESTING INPUT, OUTPUT WAVEFORM

OUTPUT TEST LOAD CIRCUITS

*200 pF for plastic 8282/8283.

WAVEFORMS

Output Delay vs. Capacitance

8284A/8284A-1
CLOCK GENERATOR AND DRIVER FOR iAPX 86, 88 PROCESSORS

- Generates the System Clock for the iAPX 86, 88 Processors:
  5 MHz, 8 MHz with 8284A
  10 MHz with 8284A-1
- Uses a Crystal or a TTL Signal for Frequency Source
- Provides Local READY and Multibus™ READY Synchronization
- 18-Pin Package
- Single +5V Power Supply
- Generates System Reset Output from Schmitt Trigger Input
- Capable of Clock Synchronization with Other 8284As

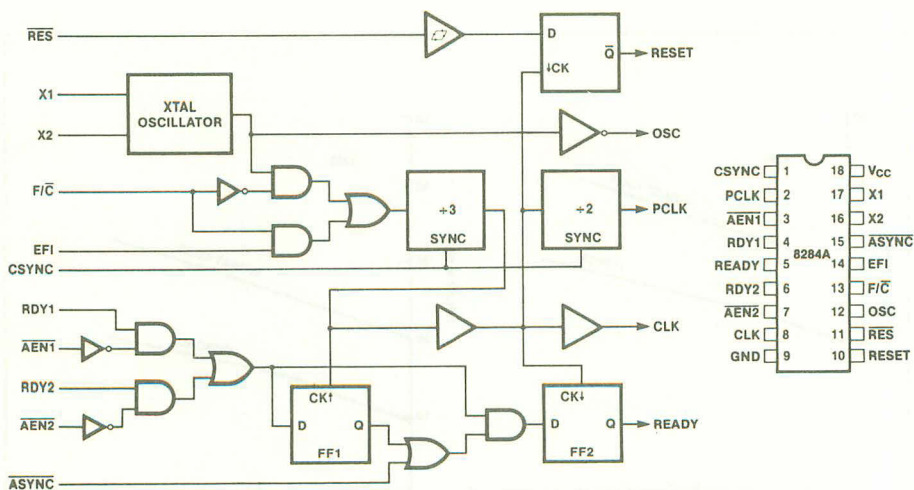



Figure 1. 8284A/8284A-1 Block Diagram

Figure 2. 8284A/8284A-1 Pin Configuration

ABSOLUTE MAXIMUM RATINGS*

Temperature Under Bias .... 0°C to 70°C
Storage Temperature .... −65°C to +150°C
All Output and Supply Voltages .... −0.5V to +7V
All Input Voltages .... −1.0V to +5.5V
Power Dissipation .... 1 Watt

**NOTICE: Stresses above those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.*

D.C. CHARACTERISTICS (T_A = 0°C to 70°C, V_{CC} = 5V ± 10%)

Symbol	Parameter	Min.	Max.	Units	Test Conditions
I_F	Forward Input Current (ASYNC) Other Inputs		−1.3 −0.5	mA mA	V_F = 0.45V V_F = 0.45V
I_R	Reverse Input Current (ASYNC) Other Inputs		50 50	μA μA	V_R = V_{CC} V_R = 5.25V
V_C	Input Forward Clamp Voltage		−1.0	V	I_C = −5mA
I_{CC}	Power Supply Current		162	mA	
V_{IL}	Input LOW Voltage		0.8	V	
V_{IH}	Input HIGH Voltage	2.0		V	
V_{IHR}	Reset Input HIGH Voltage	2.6		V	
V_{OL}	Output LOW Voltage		0.45	V	5mA
V_{OH}	Output HIGH Voltage CLK Other Outputs	4 2.4		V V	−1mA −1mA
$V_{IHR} - V_{ILR}$	RES Input Hysteresis	0.25		V	

A.C. CHARACTERISTICS (T_A = 0°C to 70°C, V_{CC} = 5V ± 10%)

TIMING REQUIREMENTS

Symbol	Parameter	Min.	Max.	Units	Test Conditions
t_{EHEL}	External Frequency HIGH Time	13		ns	90%-90% V_{IN}
t_{ELEH}	External Frequency LOW Time	13		ns	10%-10% V_{IN}
t_{ELEL}	EFI Period	33		ns	(Note 1)
	XTAL Frequency	12	25	MHz	
t_{R1VCL}	RDY1, RDY2 Active Setup to CLK	35		ns	ASYNC = HIGH
t_{R1VCH}	RDY1, RDY2 Active Setup to CLK	35		ns	ASYNC = LOW
t_{R1VCL}	RDY1, RDY2 Inactive Setup to CLK	35		ns	
t_{CLR1X}	RDY1, RDY2 Hold to CLK	0		ns	
t_{AYVCL}	ASYNC Setup to CLK	50		ns	
t_{CLAYX}	ASYNC Hold to CLK	0		ns	
t_{A1VR1V}	AEN1, AEN2 Setup to RDY1, RDY2	15		ns	
t_{CLA1X}	AEN1, AEN2 Hold to CLK	0		ns	
t_{YHEH}	CSYNC Setup to EFI	20		ns	
t_{EHYL}	CSYNC Hold to EFI	10		ns	
t_{YHYL}	CSYNC Width	$2 \cdot t_{ELEL}$		ns	
t_{I1HCL}	RES Setup to CLK	65		ns	(Note 1)
t_{CLI1H}	RES Hold to CLK	20		ns	(Note 1)
t_{ILIH}	Input Rise Time		20	ns	From 0.8V to 2.0V
t_{ILIL}	Input Fall Time		12	ns	From 2.0V to 0.8V

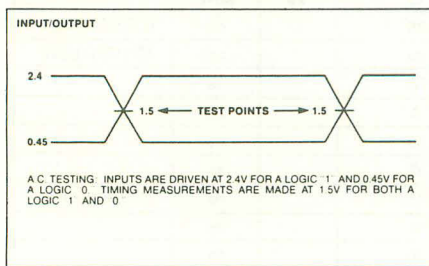
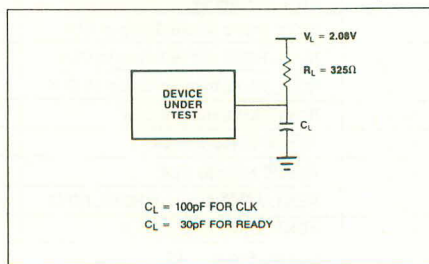
A.C. CHARACTERISTICS (Continued)

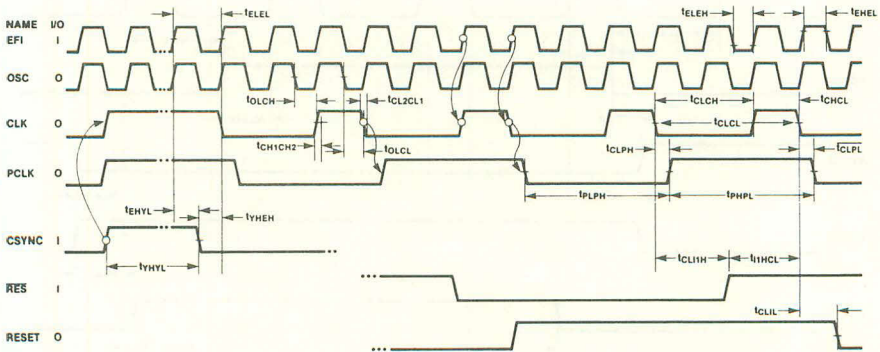
TIMING RESPONSES

Symbol	Parameter	Min. 8284A	Min. 8284A-1	Max.	Units	Test Conditions
t_{CLCL}	CLK Cycle Period	125	100		ns	
t_{CHCL}	CLK HIGH Time	$(\frac{1}{3} t_{CLCL})+2$	39		ns	
t_{CLCH}	CLK LOW Time	$(\frac{2}{3} t_{CLCL})-15$	53		ns	
t_{CH1CH2} t_{CL2CL1}	CLK Rise or Fall Time			10	ns	1.0V to 3.5V
t_{PHPL}	PCLK HIGH Time	$t_{CLCL}-20$	$t_{CLCL}-20$		ns	
t_{PLPH}	PCLK LOW Time	$t_{CLCL}-20$	$t_{CLCL}-20$		ns	
t_{RYLCL}	Ready Inactive to CLK (See Note 3)	−8	−8		ns	
t_{RYHCH}	Ready Active to CLK (See Note 2)	$(\frac{2}{3} t_{CLCL})-15$	53		ns	
t_{CLIL}	CLK to Reset Delay			40	ns	
t_{CLPH}	CLK to PCLK HIGH DELAY			22	ns	
t_{CLPL}	CLK to PCLK LOW Delay			22	ns	
t_{OLCH}	OSC to CLK HIGH Delay	−5	−5	22	ns	
t_{OLCL}	OSC to CLK LOW Delay	2	2	35	ns	
t_{OLOH}	Output Rise Time (except CLK)			20	ns	From 0.8V to 2.0V
t_{OHOL}	Output Fall Time (except CLK)			12	ns	From 2.0V to 0.8V

NOTES:

1. Setup and hold necessary only to guarantee recognition at next clock.
2. Applies only to T3 and TW states.
3. Applies only to T2 states.

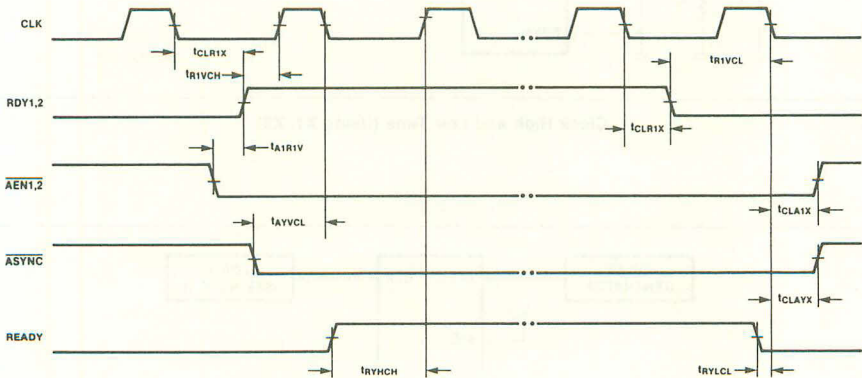
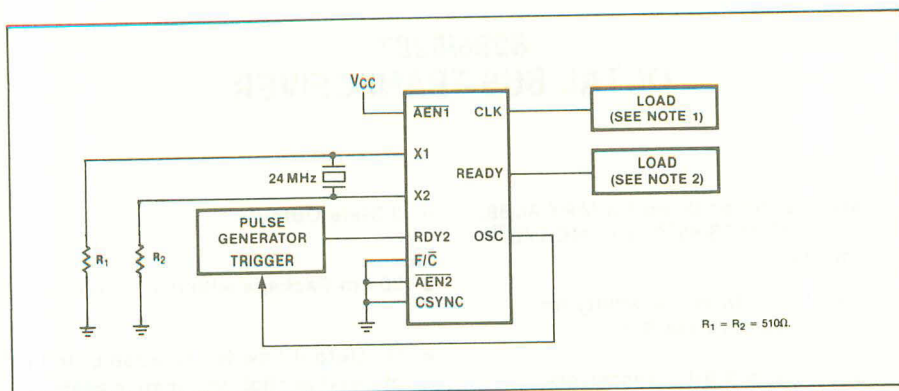
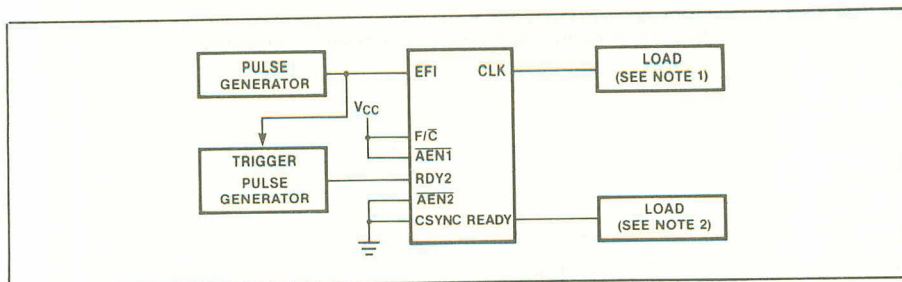
A.C. TESTING INPUT, OUTPUT WAVEFORM

A.C. TESTING LOAD CIRCUIT

WAVEFORMS

READY SIGNALS (FOR ASYNCHRONOUS DEVICES)

CLK
tCLR1X
tR1VCH
tR1VCL
RDY1,2
tCLR1X
tA1R1V
$\overline{\text{AEN1,2}}$
tAYVCL
tCLA1X
$\overline{\text{ASYNC}}$
tCLAYX
READY
tRYHCH
tRYLCL

WAVEFORMS (Continued)

Clock High and Low Time (Using X1, X2)

Clock High and Low Time (Using EFI)

Ready to Clock (Using X1, X2)

Ready to Clock (Using EFI)

NOTES:
1. C_L = 100 pF
2. C_L = 30 pF

8286/8287
OCTAL BUS TRANSCEIVER

- **Data Bus Buffer Driver for iAPX 86,88, MCS-80™, MCS-85™, and MCS-48™ Families**
- **High Output Drive Capability for Driving System Data Bus**
- **Fully Parallel 8-Bit Transceivers**
- **3-State Outputs**
- **20-Pin Package with 0.3" Center**
- **No Output Low Noise when Entering or Leaving High Impedance State**

The 8286 and 8287 are 8-bit bipolar transceivers with 3-state outputs. The 8287 inverts the input data at its outputs while the 8286 does not. Thus, a wide variety of applications for buffering in microcomputer systems can be met.

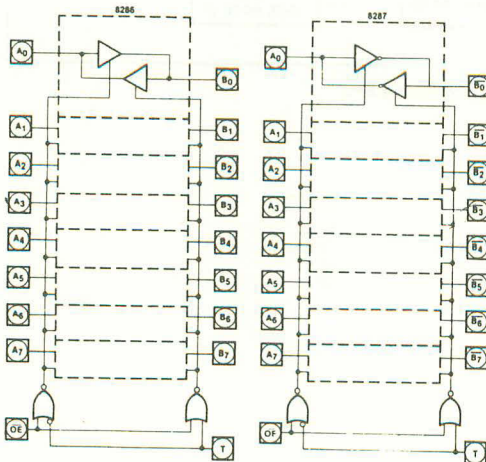



Figure 1. Logic Diagrams

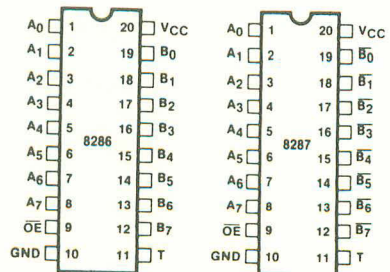
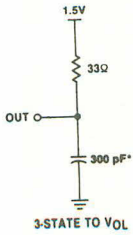
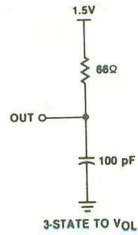
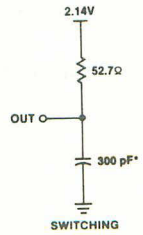
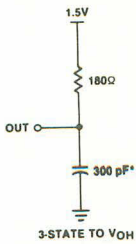
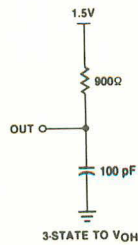
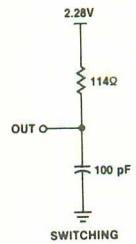



Figure 2. Pin Configurations

TEST LOAD CIRCUITS

*200 pF for plastic 8286/8287

ABSOLUTE MAXIMUM RATINGS*

Temperature Under Bias .... 0°C to 70°C
Storage Temperature .... −65°C to +150°C
All Output and Supply Voltages .... −0.5V to +7V
All Input Voltages .... −1.0V to +5.5V
Power Dissipation .... 1 Watt

**NOTICE: Stresses above those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only ahd functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.*

D.C. CHARACTERISTICS (V_{CC} = +5V ±10%, T_A = 0°C to 70°C)

Symbol	Parameter	Min	Max	Units	Test Conditions
V_C	Input Clamp Voltage		-1	V	I_C = -5 mA
I_{CC}	Power Supply Current—8287 —8286		130 160	mA mA	
I_F	Forward Input Current		-0.2	mA	V_F = 0.45V
I_R	Reverse Input Current		50	μA	V_R = 5.25V
V_{OL}	Output Low Voltage —B Outputs —A Outputs		.45 .45	V V	I_{OL} = 32 mA I_{OL} = 16 mA
V_{OH}	Output High Voltage —B Outputs —A Outputs	2.4 2.4		V V	I_{OH} = -5 mA I_{OH} = -1 mA
I_{OFF} I_{OFF}	Output Off Current Output Off Current		I_F I_R		V_{OFF} = 0.45V V_{OFF} = 5.25V
V_{IL}	Input Low Voltage —A Side —B Side		0.8 0.9	V V	V_{CC} = 5.0V, See Note 1 V_{CC} = 5.0V, See Note 1
V_{IH}	Input High Voltage	2.0		V	V_{CC} = 5.0V, See Note 1
C_{IN}	Input Capacitance		12	pF	F = 1 MHz V_{BIAS} = 2.5V, V_{CC} = 5V T_A = 25°C

NOTE:
1. B Outputs—I_{OL} = 32 mA, I_{OH} = −5 mA, C_L = 300 pF*: A Outputs—I_{OL} = 16 mA, I_{OH} = −1 mA, C_L = 100 pF.

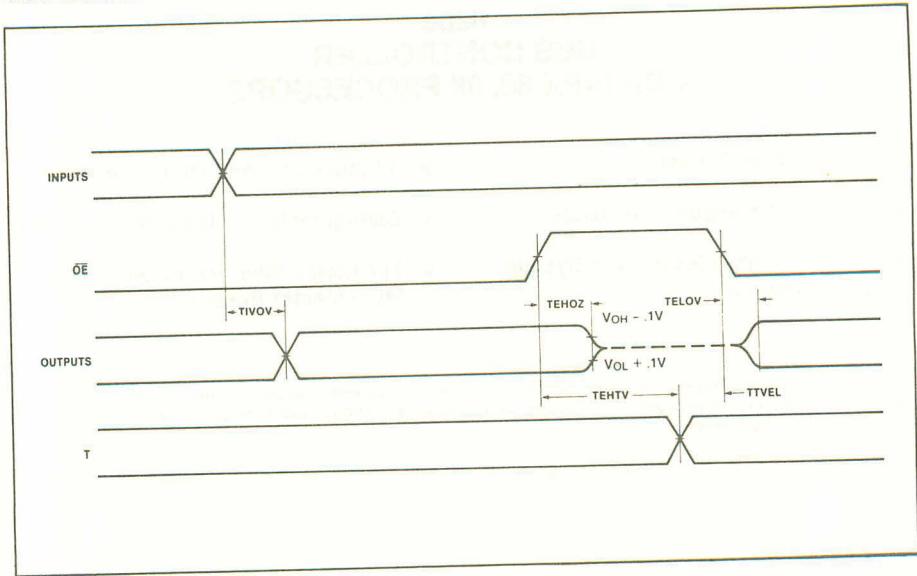
A.C. CHARACTERISTICS (V_{CC} = +5V ±10%, T_A = 0°C to 70°C)

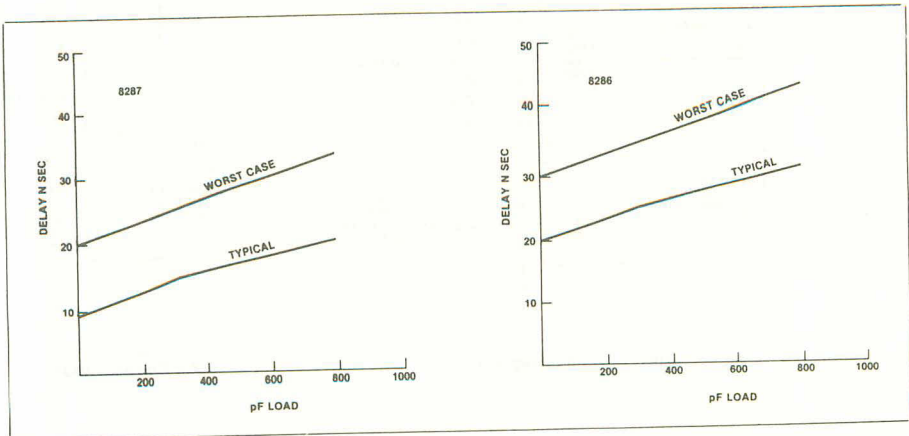
Loading: B Outputs—I_{OL} = 32 mA, I_{OH} = −5 mA, C_L = 300 pF*
A Outputs—I_{OL} = 16 mA, I_{OH} = −1 mA, C_L = 100 pF

Symbol	Parameter	Min	Max	Units	Test Conditions
TIVOV	Input to Output Delay Inverting Non-Inverting	 5 5	 22 30	 ns ns	(See Note 1)
TEHTV	Transmit/Receive Hold Time	5		ns	
TTVEL	Transmit/Receive Setup	10		ns	
TEHOZ	Output Disable Time	5	18	ns	
TELOV	Output Enable Time	10	30	ns	
TILIH, TOLOH	Input, Output Rise Time		20	ns	From 0.8 V to 2.0V
TIHIL, TOHOL	Input, Output Fall Time		12	ns	From 2.0V to 8.0V

*C_L = 200 pF for plastic 8286/8287

NOTE:
1. See waveforms and test load circuit on following page.

WAVEFORMS

NOTE:

1. All timing measurements are made at 1.5V unless otherwise noted.

Output Delay versus Capacitance

8288 BUS CONTROLLER FOR iAPX 86, 88 PROCESSORS

- **Bipolar Drive Capability**
- **Provides Advanced Commands**
- **Provides Wide Flexibility in System Configurations**
- **3-State Command Output Drivers**
- **Configurable for Use with an I/O Bus**
- **Facilitates Interface to One or Two Multi-Master Busses**

The Intel® 8288 Bus Controller is a 20-pin bipolar component for use with medium-to-large iAPX 86, 88 processing systems. The bus controller provides command and control timing generation as well as bipolar bus drive capability while optimizing system performance.

A strapping option on the bus controller configures it for use with a multi-master system bus and separate I/O bus.

Figure 1. Block Diagram

Figure 2. Pin Configuration

8288

ABSOLUTE MAXIMUM RATINGS*

Temperature Under Bias .... 0°C to 70°C
Storage Temperature .... −65°C to +150°C
All Output and Supply Voltages .... −0.5V to +7V
All Input Voltages .... −1.0V to +5.5V
Power Dissipation .... 1.5 Watt

**NOTICE: Stresses above those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.*

D.C. CHARACTERISTICS (V_{CC} = 5V ± 10%, T_A = 0°C to 70°C)

Symbol	Parameter	Min.	Max.	Unit	Test Conditions
V_C	Input Clamp Voltage		−1	V	I_C = −5 mA
I_{CC}	Power Supply Current		230	mA	
I_F	Forward Input Current		−0.7	mA	V_F = 0.45V
I_R	Reverse Input Current		50	μA	$V_R = V_{CC}$
V_{OL}	Output Low Voltage Command Outputs Control Outputs		 0.5 0.5	 V V	 I_{OL} = 32 mA I_{OL} = 16 mA
V_{OH}	Output High Voltage Command Outputs Control Outputs	 2.4 2.4		 V V	 I_{OH} = −5 mA I_{OH} = −1 mA
V_{IL}	Input Low Voltage		0.8	V	
V_{IH}	Input High Voltage	2.0		V	
I_{OFF}	Output Off Current		100	μA	V_{OFF} = 0.4 to 5.25V

A.C. CHARACTERISTICS (V_{CC} = 5V ± 10%, T_A = 0°C to 70°C)

TIMING REQUIREMENTS

Symbol	Parameter	Min.	Max.	Unit	Test Conditions
TCLCL	CLK Cycle Period	100		ns	
TCLCH	CLK Low Time	50		ns	
TCHCL	CLK High Time	30		ns	
TSVCH	Status Active Setup Time	35		ns	
TCHSV	Status Active Hold Time	10		ns	
TSHCL	Status Inactive Setup Time	35		ns	
TCLSH	Status Inactive Hold Time	10		ns	
TILIH	Input, Rise Time		20	ns	From 0.8V to 2.0V
TIHIL	Input, Fall Time		12	ns	From 2.0V to 0.8V

A.C. CHARACTERISTICS (Continued)

TIMING RESPONSES

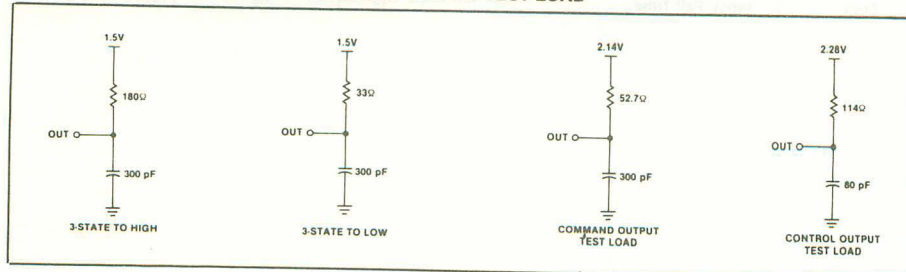
Symbol	Parameter	Min.	Max.	Unit	Test Conditions
TCVNV	Control Active Delay	5	45	ns	
TCVNX	Control Inactive Delay	10	45	ns	
TCLLH, TCLMCH	ALE MCE Active Delay (from CLK)		20	ns	
TSVLH, TSVMCH	ALE MCE Active Delay (from Status)		20	ns	
TCHLL	ALE Inactive Delay	4	15	ns	$\overline{MRDC}$
TCLML	Command Active Delay	10	35	ns	$\overline{IORC}$
TCLMH	Command Inactive Delay	10	35	ns	$\overline{MWTC}$ $I_{OL} = 32$ mA
TCHDTL	Direction Control Active Delay		50	ns	$\overline{IOWC}$ $I_{OH} = -5$ mA
TCHDTH	Direction Control Inactive Delay		30	ns	$\overline{INTA}$ $C_L = 300$ pF
TAELCH	Command Enable Time		40	ns	$\overline{AMWC}$
TAEHCZ	Command Disable Time		40	ns	$\overline{AIOWC}$
TAELCV	Enable Delay Time	115	200	ns	$I_{OL} = 16$ mA
TAEVNV	AEN to DEN		20	ns	Other $I_{OH} = -1$ mA
TCEVNV	CEN to DEN, PDEN		25	ns	$C_L = 80$ pF
TCELRH	CEN to Command		TCLML	ns	
TOLOH	Output, Rise Time		20	ns	From 0.8V to 2.0V
TOHOL	Output, Fall Time		12	ns	From 2.0V to 0.8V

A.C. TESTING INPUT, OUTPUT WAVEFORM

TEST LOAD CIRCUITS—3-STATE COMMAND OUTPUT TEST LOAD

WAVEFORMS

NOTES:
1. ADDRESS/DATA BUS IS SHOWN ONLY FOR REFERENCE PURPOSES.
2. LEADING EDGE OF ALE AND MCE IS DETERMINED BY THE FALLING EDGE OF CLK OR STATUS GOING ACTIVE, WHICHEVER OCCURS LAST.
3. ALL TIMING MEASUREMENTS ARE MADE AT 1.5V UNLESS SPECIFIED OTHERWISE.

WAVEFORMS (Continued)

8289
BUS ARBITER

- **Provides Multi-Master System Bus Protocol**
- **Synchronizes iAPX 86, 88 Processors with Multi-Master Bus**
- **Provides Simple Interface with 8288 Bus Controller**
- **Four Operating Modes for Flexible System Configuration**
- **Compatible with Intel Bus Standard MULTIBUS™**
- **Provides System Bus Arbitration for 8089 IOP in Remote Mode**

The Intel 8289 Bus Arbiter is a 20-pin, 5-volt-only bipolar component for use with medium to large iAPX 86, 88 multi-master/multiprocessing systems. The 8289 provides system bus arbitration for systems with multiple bus masters, such as an 8086 CPU with 8089 IOP in its REMOTE mode, while providing bipolar buffering and drive capability.

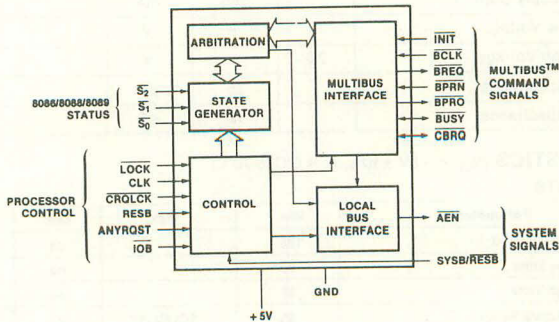



Figure 1. Block Diagram

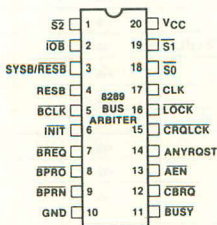



Figure 2. Pin Diagram

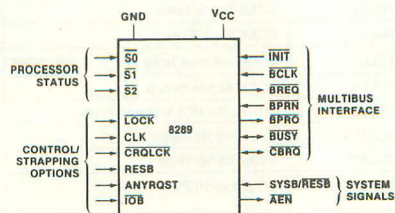



Figure 3. Functional Pinout

ABSOLUTE MAXIMUM RATINGS*

Temperature Under Bias ....0°C to 70°C
Storage Temperature.............. − 65°C to + 150°C
All Output and Supply Voltages........ − 0.5V to + 7V
All Input Voltages.................. − 1.0V to + 5.5V
Power Dissipation.........................1.5 Watt

**NOTICE: Stresses above those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.*

D.C. CHARACTERISTICS (T_A = 0°C to 70°C, V_{CC} = +5V ±10%)

Symbol	Parameter	Min.	Max.	Units	Test Condition
V_C	Input Clamp Voltage		− 1.0	V	V_{CC} = 4.50V, I_C = − 5 mA
I_F	Input Forward Current		− 0.5	mA	V_{CC} = 5.50V, V_F = 0.45V
I_R	Reverse Input Leakage Current		60	μA	V_{CC} = 5.50, V_R = 5.50
V_{OL}	Output Low Voltage $\overline{BUSY}$, $\overline{CBRQ}$ $\overline{AEN}$ $\overline{BPRO}$, $\overline{BREQ}$		 0.45 0.45 0.45	 V V V	 I_{OL} = 20 mA I_{OL} = 16 mA I_{OL} = 10 mA
V_{OH}	Output High Voltage $\overline{BUSY}$, $\overline{CBRQ}$	Open Collector			
	All Other Outputs	2.4		V	I_{OH} = 400 μA
I_{CC}	Power Supply Current		165	mA	
V_{IL}	Input Low Voltage		.8	V	
V_{IH}	Input High Voltage	2.0		V	
Cin Status	Input Capacitance		25	pF	
Cin (Others)	Input Capacitance		12	pF	

A.C. CHARACTERISTICS (V_{CC} = +5V ±10%, T_A = 0°C to 70°C)

TIMING REQUIREMENTS

Symbol	Parameter	Min.	Max.	Unit	Test Condition
TCLCL	CLK Cycle Period	125		ns	
TCLCH	CLK Low Time	65		ns	
TCHCL	CLK High Time	35		ns	
TSVCH	Status Active Setup	65	TCLCL-10	ns	
TSHCL	Status Inactive Setup	50	TCLCL-10	ns	
THVCH	Status Active Hold	10		ns	
THVCL	Status Inactive Hold	10		ns	
TBYSBL	BUSY↑↓Setup to BCLK↓	20		ns	
TCBSBL	CBRQ↑↓Setup to BCLK↓	20		ns	
TBLBL	BCLK Cycle Time	100		ns	
TBHCL	BCLK High Time	30	.65[TBLBL]	ns	
TCLLL1	LOCK Inactive Hold	10		ns	
TCLLL2	LOCK Active Setup	40		ns	
TPNBL	BPRN↓↑to BCLK Setup Time	15		ns	
TCLSR1	SYSB/RESB Setup	0		ns	
TCLSR2	SYSB/RESB Hold	20		ns	
TIVIH	Initialization Pulse Width	3 TBLBL+ 3 TCLCL		ns	
TILIH	Input Rise Time		20	ns	From 0.8 to 2.0V
TIHIL	Input Fall Time		12	ns	From 2.0V to 0.8V

A.C. CHARACTERISTICS (Continued)

TIMING RESPONSES

Symbol	Parameter	Min.	Max.	Unit	Test Condition
TBLBRL	BCLK to BREQ Delay↓↑		35	ns	
TBLPOH	BCLK to BPRO↓↑ (See Note 1)		40	ns	
TPNPO	BPRN↓↑to BPRO↓↑Delay (See Note 1)		25	ns	
TBLBYL	BCLK to BUSY Low		60	ns	
TBLBYH	BCLK to BUSY Float (See Note 2)		35	ns	
TCLAEH	CLK to AEN High		65	ns	
TBLAEL	BCLK to AEN Low		40	ns	
TBLCBL	BCLK to CBRQ Low		60	ns	
TRLCRH	BCLK to CBRQ Float (See Note 2)		35	ns	
TOLOH	Output Rise Time		20	ns	From 0.8V to 2.0V
TOHOL	Output Fall Time		12	ns	From 2.0V to 0.8V

↓↑ Denotes that spec applies to both transitions of the signal.

NOTES:
1. BCLK generates the first BPRO wherein subsequent BPRO changes lower in the chain are generated through BPRON.
2. Measured at .5V above GND.

A.C. TESTING INPUT, OUTPUT WAVEFORM

A.C. TESTING LOAD CIRCUIT

WAVEFORMS

NOTES:
1. LOCK ACTIVE CAN OCCUR DURING ANY STATE, AS LONG AS THE RELATIONSHIPS SHOWN ABOVE WITH RESPECT TO THE CLK ARE MAINTAINED. LOCK INACTIVE HAS NO CRITICAL TIME AND CAN BE ASYNCHRONOUS. CRQLCK HAS NO CRITICAL TIMING AND IS CONSIDERED AN ASYNCHRONOUS INPUT SIGNAL
2. GLITCHING OF SYSB/RESB PIN IS PERMITTED DURING THIS TIME. AFTER ϕ 2 OF T1, AND BEFORE ϕ1 OF T4, SYSB/RESB SHOULD BE STABLE.
3. AEN LEADING EDGE IS RELATED TO BCLK, TRAILING EDGE TO CLK. THE TRAILING EDGE OF AEN OCCURS AFTER BUS PRIORITY IS LOST.

ADDITIONAL NOTES:

The signals related to CLK are typical processor signals, and do not relate to the depicted sequence of events of the signals referenced to BCLK. The signals shown related to the BCLK represent a hypothetical sequence of events for illustration. Assume 3 bus arbiters of priorities 1, 2 and 3 configured in serial priority resolving scheme as shown in Figure 6. Assume arbiter 1 has the bus and is holding busy low. Arbiter #2 detects its processor wants the bus and pulls low BREQ#2. If BPRN#2 is high (as shown), arbiter #2 will pull low CBRQ line. CBRQ signals to the higher priority arbiter #1 that a lower priority arbiter wants the bus. [A higher priority arbiter would be granted BPRN when it makes the bus request rather than having to wait for another arbiter to release the bus through CBRQ].** Arbiter #1 will relinquish the multi-master system bus when it enters a state not requiring it (see Table 1), by lowering its BPRO#1 (tied to BPRN#2) and releasing BUSY. Arbiter #2 now sees that it has priority from BPRN#2 being low and releases CBRQ. As soon as BUSY signifies the bus is available (high), arbiter #2 pulls BUSY low on next falling edge of BCLK. Note that if arbiter #2 didn't want the bus at the time it received priority, it would pass priority to the next lower priority arbiter by lowering its BPRO #2 [TPNPO].

**Note that even a higher priority arbiter which is acquiring the bus through BPRN will momentarily drop CBRQ until it has acquired the bus.

8089
8 & 16-BIT HMOS I/O PROCESSOR

- **High Speed DMA Capabilities Including I/O to Memory, Memory to I/O, Memory to Memory, and I/O to I/O**
- **iAPX 86, 88 Compatible: Removes I/O Overhead from CPU in iAPX 86/11 or 88/11 Configuration**
- **Allows Mixed Interface of 8- & 16-Bit Peripherals, to 8- & 16-Bit Processor Busses**
- **1 Mbyte Addressability**
- **Memory Based Communication with CPU**
- **Supports LOCAL or REMOTE I/O Processing**
- **Flexible, Intelligent DMA Functions Including Translation, Search, Word Assembly/Disassembly**
- **MULTIBUS™ Compatible System Interface**

The Intel® 8089 is a revolutionary concept in microprocessor input/output processing. Packaged in a 40-pin DIP package, the 8089 is a high performance processor implemented in N-channel, depletion load silicon gate technology (HMOS). The 8089's instruction set and capabilities are optimized for high speed, flexible and efficient I/O handling. It allows easy interface of Intel's 16-bit iAPX 86 and 8-bit iAPX 88 microprocessors with 8- and 16-bit peripherals. In the REMOTE configuration, the 8089 bus is user definable allowing it to be compatible with any 8/16-bit Intel microprocessor, interfacing easily to the Intel multiprocessor system bus standard MULTIBUS™.

The 8089 performs the function of an intelligent DMA controller for the Intel iAPX 86, 88 family and with its processing power, can remove I/O overhead from the iAPX 86 or iAPX 88. It may operate completely in parallel with a CPU, giving dramatically improved performance in I/O intensive applications. The 8089 provides two I/O channels, each supporting a transfer rate up to 1.25 mbyte/sec at the standard clock frequency of 5 MHz. Memory based communication between the IOP and CPU enhances system flexibility and encourages software modularity, yielding more reliable, easier to develop systems.

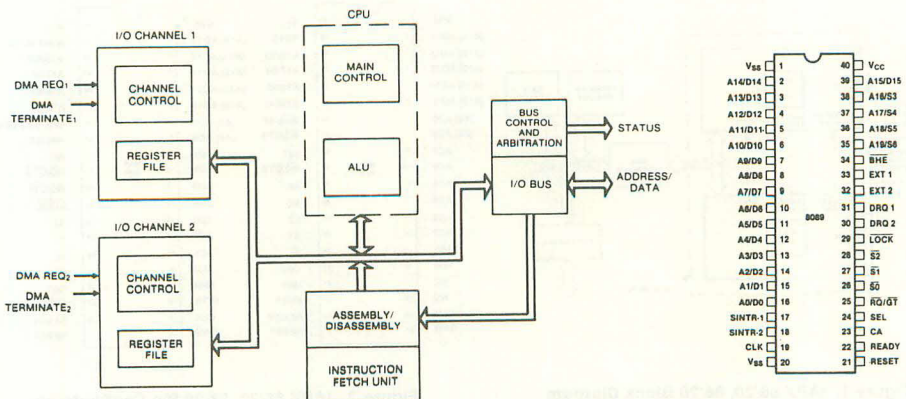



Figure 1. 8089 I/O Processor Block Diagram

Figure 2. 8089 Pin Configuration

iAPX 86/20
iAPX 88/20
NUMERIC DATA PROCESSOR

- **High Performance 2-Chip Numeric Data Processor**
- **Standard iAPX 86/10, 88/10 Instruction Set Plus Arithmetic, Trigonometric, Exponential, and Logarithmic Instructions For All Data Types**
- **All 24 iAPX 86/10, 88/10 Addressing Modes Available**
- **Conforms To Proposed IEEE Floating Point Standard**
- **Support 8 Data Types: 8-, 16-, 32-, 64-Bit Integers, 32-, 64-, 80-Bit Floating Point, and 18-Digit BCD Operands**
- **8x80-Bit Individually Addressable Register Stack plus 14 General Purpose Registers**
- **7 Built-in Exception Handling Functions**
- **MULTIBUS System Compatible Interface**

The Intel iAPX 86/20 and iAPX 88/20 are two-chip numeric data processors (NDP's). They provide the instructions and data types needed for high-performance numeric applications. The NDP provides 100 times the performance of an iAPX 86/10, 88/10 CPU alone for numeric processing. The iAPX 86/20 consists of an iAPX 86/10 (16-bit 8086 CPU) and a numeric processor extension (NPX), the 8087. The iAPX 88/20 consists of the NPX in conjunction with the iAPX 88/10 (8-bit 8088 CPU). The NDP conforms to the proposed IEEE Floating Point Standard.

Both components of the iAPX 86/20 and iAPX 88/20 are implemented in N-channel, depletion load, silicon gate technology (HMOS), housed in two 40-pin packages. The iAPX 86/20, 88/20 adds 68 numeric processing instructions to the iAPX 86/10, 88/10 instruction set and eight 80-bit registers to the register set.

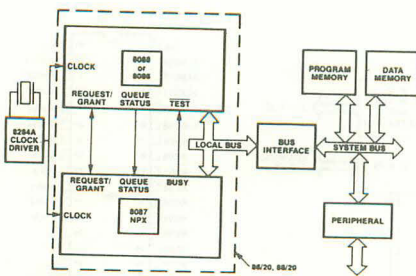



Figure 1. iAPX 86/20, 88/20 Block Diagram

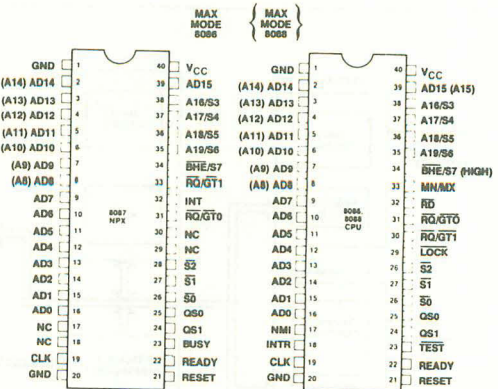



Figure 2. iAPX 86/20, 88/20 Pin Configuration

JULY 1981

iAPX 86/30
iAPX 88/30
OPERATING SYSTEM PROCESSORS

80130-3

- **High-Performance 2-Chip Data Processors Containing Operating System Primitives**
- **Standard iAPX 86/10, 88/10 Instruction Set Plus Task Management, Interrupt Management, Message Passing, Synchronization and Memory Allocation Primitives**
- **Fully Extendable To and Compatible With iRMX 86**
- **Supports Five Operating System Data Types: Jobs, Tasks, Segments, Mailboxes, Regions**
- **35 Operating System Primitives**
- **Built-In Operating System Timers and Interrupt Control Logic Expandable From 8 to 57 Interrupts**
- **8086/8087/8088 Compatible At Up To 8 MHz Without Wait States**
- **MULTIBUS System Compatible Interface**

The Intel iAPX 86/30 and iAPX 88/30 are two-chip microprocessors offering general-purpose CPU (8086) instructions combined with real-time operating system support. They provide a foundation for multiprogramming and multitasking applications. The iAPX 86/30 consists of an iAPX 86/10 (16-bit 8086 CPU) and an Operating System Firmware (OSF) component (80130). The 88/30 consists of the OSF and an iAPX 88/10 (8-bit 8088 CPU).

Both components of the 86/30 and 88/30 are implemented in N-channel, depletion-load, silicon-gate technology (HMOS), and are housed in 40-pin packages. The 86/30 and 88/30 provide all the functions of the iAPX 86/10, 88/10 processors plus 35 operating system primitives, hardware support for eight interrupts, a system timer, a delay timer and a baud rate generator.

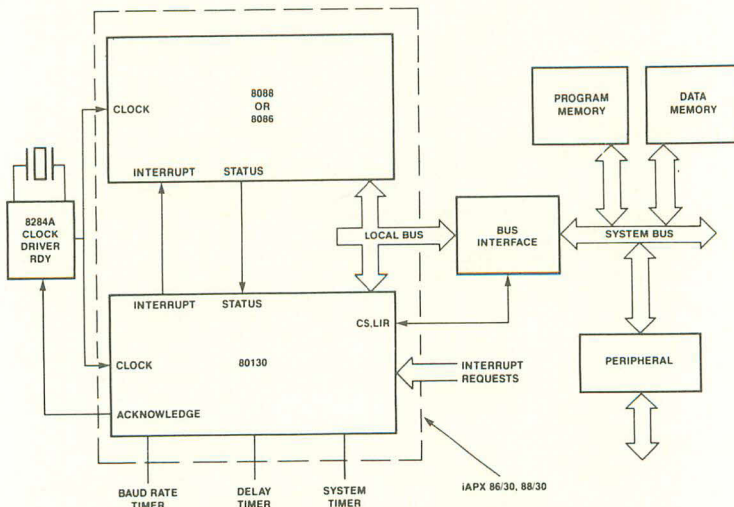



Figure 1. iAPX 86/30, 88/30 Block Diagram

付録D

8088CPUについて

8088は，8ビット・データ・バスの8086マイクロプロセッサである．それ以外では，両者は同一である．したがって，以下の文では8088と8086の差について述べる．

D.1　8088のプログラム可能レジスタとアドレッシング・モード

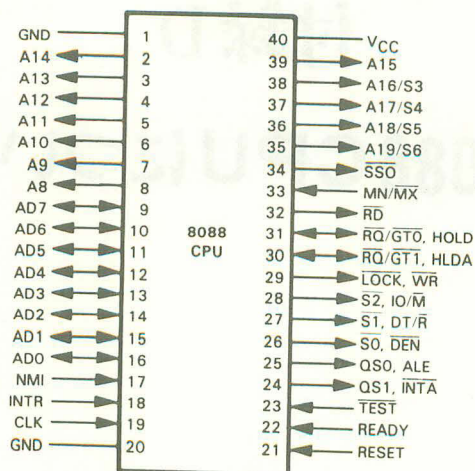
8088のプログラム可能レジスタとアドレッシング・モードは，すべてにおいて8086と同一である．実行速度を除いて，プログラマにとって8088は8086と同じである．

D.2　8088CPUのピンと信号

8088CPUのピンと信号を図D-1に示す（次ページ）．図10-1に示されている8086のピンと信号との比較を行なうと，ピン34だけが異なる（ピン2-8と39がアドレスだけであることを除く）．

8086では，ピン34は$\overline{\text{BHE}}$を出力する．この信号は，16ビットの8086データ・バス上の上位バイトと下位バイトとの間の区別を行なう．8088は8ビットのデータ・バスを有するので，$\overline{\text{BHE}}$と関連したロジックは無関係となる．8088は，マキシマム・モードの$\overline{\text{S0}}$ステータスをピン34（$\overline{\text{SSO}}$）に出力する．

IO/$\overline{\text{M}}$の信号は，8086と比べると，8088では反対の極性を持つ．これにより，8088は8085との互換性を有する．IO／$\overline{\text{M}}$，DT／$\overline{\text{R}}$，$\overline{\text{SSO}}$を組み合わせて，8088バス・サイクルは次のようにデコードされる．

IO／$\overline{\text{M}}$	DT／$\overline{\text{R}}$	$\overline{\text{SSO}}$	
0	0	0	コード・セグメント・アクセス
0	0	1	メモリ・リード
0	1	0	メモリ・ライト
0	1	1	ノー・オペレーション
1	0	0	インタラプト・アクノリッジ
1	0	1	I／Oリード
1	1	0	I／Oライト
1	1	1	ホルト

ピンの名前	種　　　類	型
AD0-AD 7	アドレス/データ・バス	双方向，トライステート
A 8 - A15	アドレス・バス	出力，トライステート
A16/S 3、A17/S 4	アドレス/セグメント識別子	出力，トライステート
A18/S 5	アドレス/インタラプト・イネーブル・ステータス	出力，トライステート
A19/S 6	アドレス/ステータス	出力，トライステート
$\overline{\text{SSO}}$	ステータス出力	出力，トライステート
$\overline{\text{RD}}$	リード・コントロール	出力，トライステート
READY	ウエート・ステート・リクエスト	入力
$\overline{\text{TEST}}$	テスト・コントロールのウエート	入力
INTR	インタラプト・リクエスト	入力
NMI	ノンマスカブル・インタラプト・リクエスト	入力
RESET	システム・リセット	入力
CLK	システム・クロック	入力
MN/$\overline{\text{MX}}$	マキシマム・システムではGND	
$\overline{\text{S0}}$, $\overline{\text{S1}}$, $\overline{\text{S2}}$	マシーン・サイクル・ステータス	出力，トライステート
$\overline{\text{RQ}}$/$\overline{\text{GT0}}$, $\overline{\text{RQ}}$/$\overline{\text{GT1}}$	ローカル・バス・プライオリティ・コントロール	双方向
QS0，QS 1	インストラクション・キュー・ステータス	出力
$\overline{\text{LOCK}}$	バス・ホールド・コントロール	出力，トライステート
MN/$\overline{\text{MX}}$	ミニマム・システムではVcc	
IO/$\overline{\text{M}}$	メモリまたはI/Oのアクセス	出力，トライステート
$\overline{\text{WR}}$	ライト・コントロール	出力，トライステート
ALE	アドレス・ラッチ・イネーブル	出力
DT/$\overline{\text{R}}$	データ・トランスミット/レシーブ	出力，トライステート
$\overline{\text{DEN}}$	データ・イネーブル	出力，トライステート
$\overline{\text{INTA}}$	インタラプト・アクノリッジ	出力，トライステート
HOLD	ホールド・リクエスト	入力
HLDA	ホールド・アクノリッジ	出力
Vcc，GND	パワー，グランド	

（濃い網掛け）マキシマム・システムの信号　　（薄い網掛け）ミニマム・システムの信号

図D－1　8088のピンと信号の割当て

8088は$\overline{\mathrm{BHE}}$信号を持たず，またそのような信号を必要としないので，8086についての外部メモリ・アドレッシングと$\overline{\mathrm{BHE}}$の議論は8088には適用されない．

D.3 8088のタイミングと命令実行

8088は4バイトの命令オブジェクト・コードのキューを持つ．一方8086は，6バイトの命令オブジェクト・コードのキューを持つ．8088は，1バイトまたはそれ以上キューが空になるとただちに，4バイトのキューを満たすために，命令フェッチのバス・サイクルの実行を開始する．対照的に，8086は，6バイトのキューの2バイトまたはそれ以上が空になるまで，命令オブジェクト・コード・バイトのプリフェッチを開始しない．それ以外は，8086についてのバス・サイクルとキューのロジックの記述はそのまま8088に適用できる．

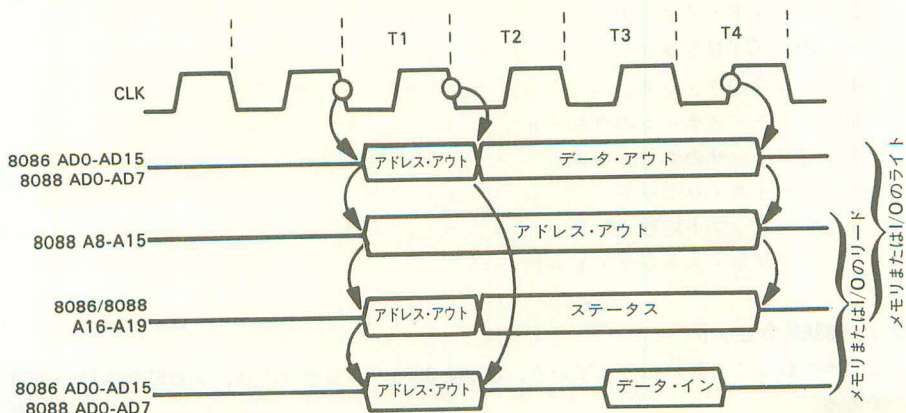
8088のキューは短いので，8086のキューではすべて含まれていた命令は，8088ではさらに命令のバイトを得るためにコードのフェッチを必要とする．各命令のフェッチには，4つのクロック・サイクルが付加されねばならない．たとえば，

```
SUB   TABLE〔BX〕,300
```

は，2バイトのディスプレイスメント(TABLE)と2バイトのイミディエイト・データ(300)を含む6バイトの命令を表わす．最初の4バイトが8088の命令キューに含まれているとすると，イミディエイト・データはそれでもフェッチされなければならず，命令の実行時間に8つのクロック・サイクルが付加される．この規則は，8088の実行時間を8086の値から得るために適用される．

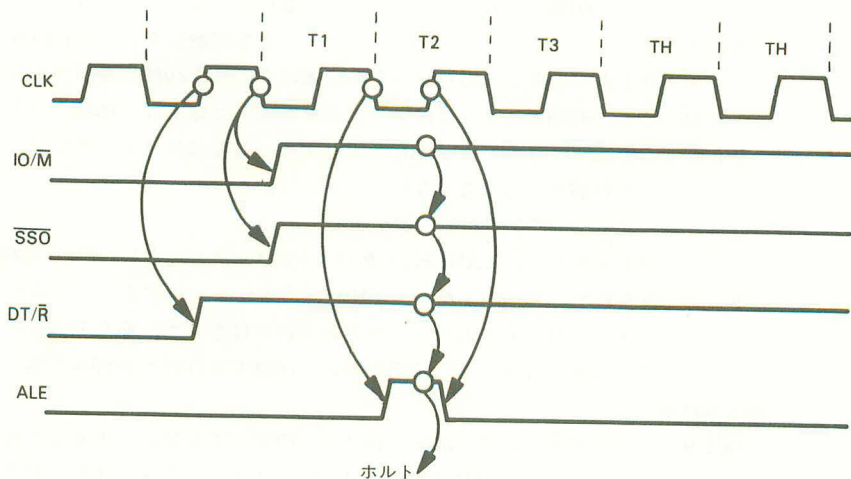
さらに，8088は8ビットのバスなので，8086が16ビットのデータをフェッチするために1つのバス・サイクルを実行していた場合は常に，2つのバス・サイクルが実行されなければならない．付録Aには，8086の実行時間が示されている．

D.4 8088のメモリとI／O素子のアクセスのバス・サイクル

8088と8086のバス・サイクルのタイミングは，多重化データ／アドレス・バス・サイクルの点だけ異なっている．タイミングの相違は8つのアドレス・バス・ラインA8－A15

に限定され，前ページの図のように示される．

8088は$\overline{BHE}$信号を持たない事実は別として，データ／アドレス・バス以外の信号のタイミングはすべて，8086と8088では同一である．

D.5 8088のホルト・ステート

ミニマム・モードでの動作は，8088のALEパルスは8086のタイミングと比較して1クロック期間だけ遅れる．これは次のように図示される．

それ以外，ホルト・ステートのロジックとタイミングは8086と8088で同一である．

D.6 8086と互換性のある8088の他のロジック

8086と8088のロジックは，次のステートとロジックで完全に同一である．

1. ウエート・ステート
2. ホールド・ステート
3. $\overline{RQ}$／$\overline{GT}$ロジック
4. ロックのロジック
5. テスト・ステートのウエート
6. プロセッサのエスケープ
7. デバイス・リセット
8. インタラプト処理
9. シングル・ステップ・モード

D.7 8088命令セット

この本の数多くの表に示されている，8086と8088の命令セットは，実行時間を除いて同一である．

欧 文 索 引

〔A〕

〔B〕

〔C〕

〔D〕

〔E〕

〔H〕

〔I〕

〔J〕

〔L〕

〔M・N〕

〔O・P・Q〕

和文索引

〔ア 行〕

〔カ 行〕

〔サ 行〕

〔タ 行〕

〔ナ 行〕

〔ハ 行〕

〔マ 行〕

〔ラ・ヤ行〕

<著者紹介>

ラッセル・レクター (Russell Rector)

1968年より計算機処理に携わる．カリフォルニア大学よりコンピュータ・サイエンスの学士号を受けた後，いくつかの重要なソフトウエア・システムの創作に参加し，後にOsborneのテクニカル・スタッフに加わり，ソフトウエア設計とOsborneの出版物の著作協力に時間をさいている．

ジョージ・アレクシー (George Alexy)

1977年にIntelに入る．8086，8088，8089，8087を含む8ビットと16ビットのマイクロプロセッサをカバーする，マイクロプロセッサの製品のアプリケーション・マネージャである．彼のグループは，シングルとマルチのプロセッサ・システムに対するシステム設計法，CPUとシステムの構成と性能に関連するリソース分配と機能的な分割に関係している．

<訳者紹介>

吉川敏則（よしかわ　としのり）

昭和46年東京工業大学工学部電子工学科卒業／昭和51年同大学院博士課程修了．工学博士／昭和52年埼玉大学工学部電子工学科講師／現在，長岡技術科学大学助教授，中央大学非常勤講師／専門はディジタル信号処理，回路解析，コンピュータのソフトウエア応用．

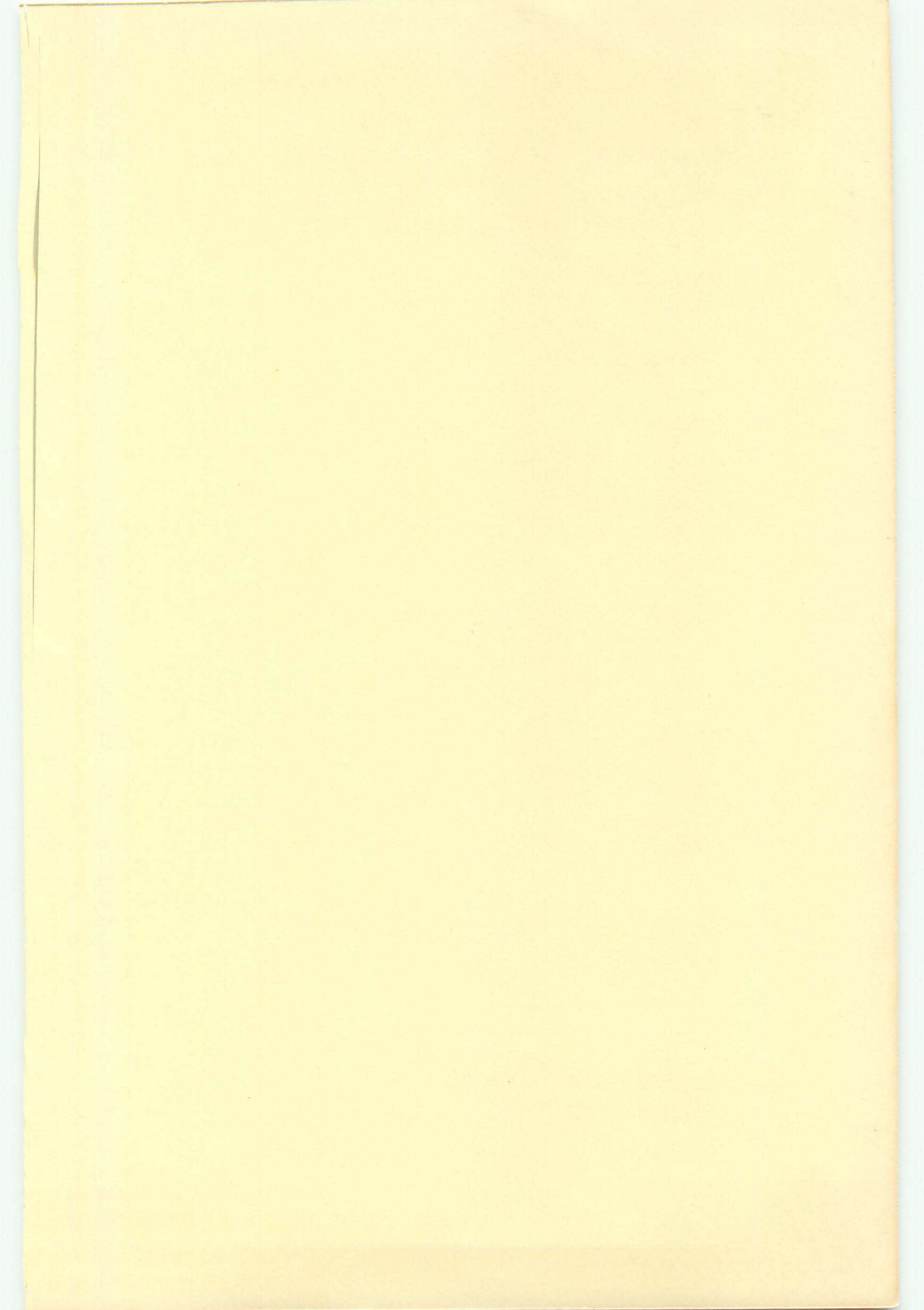
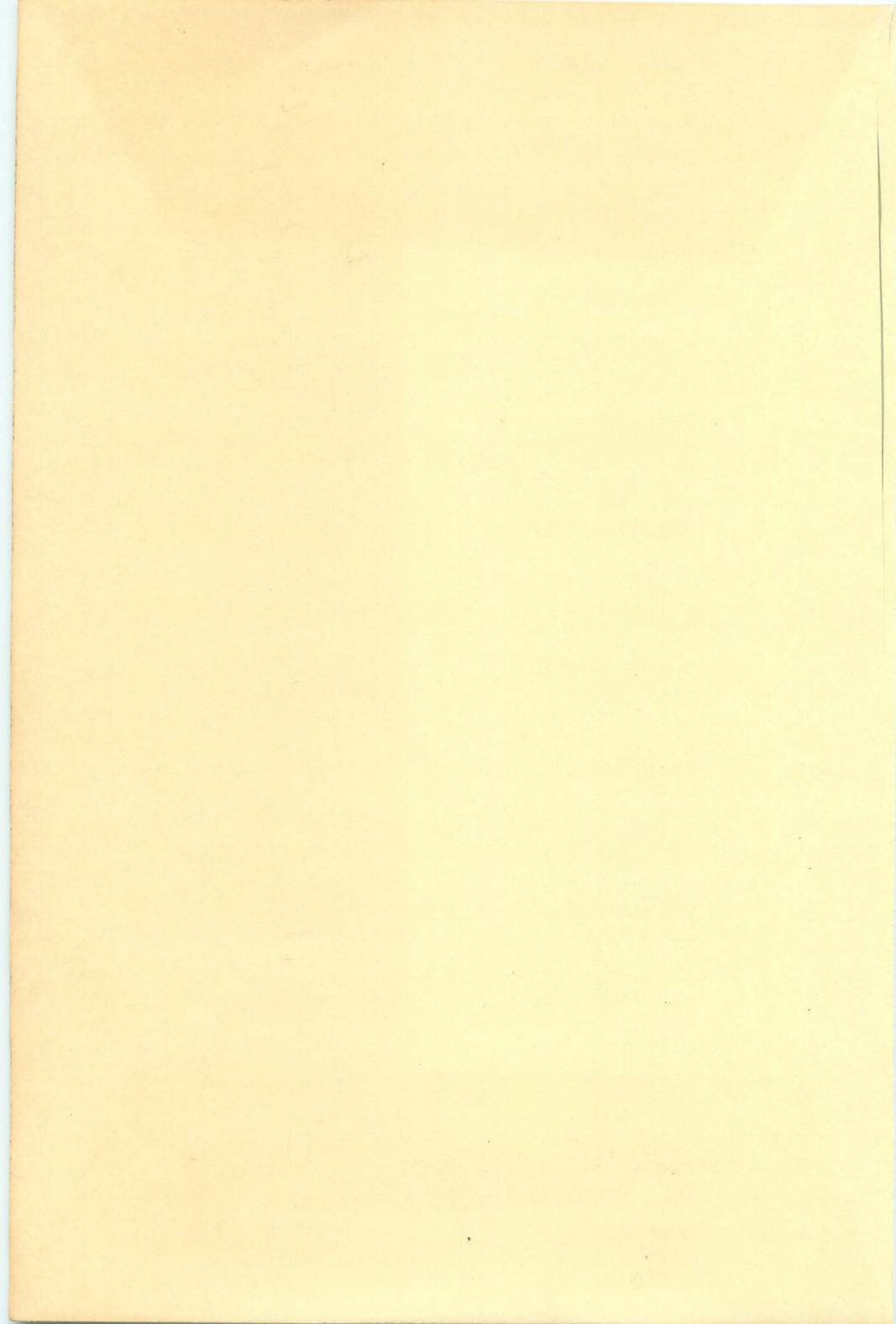
主な著書：「実用ＢＡＳＩＣ－科学技術計算」(昭晃堂)，「最新マイコン・ガイド」(日本理工出版会)，「16ビットパソコン・ガイド」(日本理工出版会)，「工学における数値計算法」(日本理工出版会)，「PC-8801実用数値計算プログラム集」(昭晃堂)，「PC-9801実用数値計算プログラム集」(昭晃堂)，「BASICノート」(日本理工出版会)，PC-9801E／F／M16ビットアセンブラ入門」(産業報知センター)

ザ8086ブック

定価4,800円

1982年9月20日　初版発行
1985年5月10日　9版発行
1986年1月25日　新装版発行

著者　R. レクター
　　　G. アレクシー
訳者　吉川敏則
発行者　三好哲雄
発行所　秋葉出版株式会社
東京都千代田区神田和泉町1－1（郵便番号101）
電話 03 (866) 5491 代表　振替 東京7-161626

印刷＝株式会社廣済堂
製本＝秋元製本株式会社

ISBN4-87184-040-9

ザ8086ブック

16ビット・マイクロプロセッサ 8086・8088の使い方

ラッセル・レクター　ジョージ・アレクシー共著　吉川敏則訳

16ビットの決定版8086

16ビット・マイクロプロセッサの本格的な応用は今後ますます盛んになるだろう。これまでの8ビット汎用マイクロプロセッサにない高度な機能は，新しい世代のマイクロコンピュータの製品を生み出す有力な武器を提供している。

定価4,800円
ISBN4-87184-040-9 C3055 ¥4800E